Parallel Performance of Molecular Dynamics Trajectory Analysis

Mahzad Khoshlessan[a], Ioannis Paraskevakos[b], Geoffrey C. Fox[c], Shantenu Jha[b], Oliver Beckstein[a,d,*]

[a]*Department of Physics, Arizona State University, Tempe, AZ 85281, USA*
[b]*Department of Electrical & Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA*
[c]*Digital Science Center, Indiana University, Bloomington, IN 47405*
[d]*Center for Biological Physics, Arizona State University, Tempe, AZ 85281, USA*

Abstract

The performance of biomolecular molecular dynamics (MD) simulations has steadily increased on modern high performance computing (HPC) resources but acceleration of the analysis of the output trajectories has lagged behind so that analyzing simulations is increasingly becoming a bottleneck. To close this gap, we studied the performance of parallel trajectory analysis with MPI and the Python *MDAnalysis* library on three different XSEDE supercomputers where trajectories were read from a Lustre parallel file system. We found that strong scaling performance was impeded by stragglers, MPI processes that were slower than the typical process and that therefore dominated the overall run time. Stragglers were less prevalent for compute-bound workloads, thus pointing to file reading as a crucial bottleneck for scaling. However, a more complicated picture emerged in which both the computation and the ingestion of data exhibited close to ideal strong scaling behavior whereas stragglers were primarily caused by either large MPI communication costs or long times to open the single shared trajectory file. We improved overall strong scaling performance by two different approaches to file access, namely subfiling (splitting the trajectory into as many trajectory segments as number of processes) and MPI-IO with Parallel HDF5 trajectory files. Applying these strategies, we obtained near ideal strong scaling on up to 384 cores (16 nodes).

*Corresponding author
Email addresses: `mkhoshle@asu.edu` (Mahzad Khoshlessan), `i.paraskev@rutgers.edu` (Ioannis Paraskevakos), `gcf@indiana.edu` (Geoffrey C. Fox), `shantenu.jha@rutgers.edu` (Shantenu Jha), `oliver.beckstein@asu.edu` (Oliver Beckstein)

We summarize our lessons-learned in guidelines and strategies on how to take advantage of the available HPC resources to gain good scalability and potentially reduce trajectory analysis times by two orders of magnitude compared to the prevalent serial approach.

Keywords: Python, MPI, HPC, MDAnalysis, MPI I/O, Global Arrays, HDF5, Straggler, Molecular Dynamics, Big Data, Trajectory Analysis

1. Introduction

Molecular dynamics (MD) simulations are a powerful method to generate new insights into the function of biomolecules [1–5]. These simulations produce trajectories—time series of atomic coordinates—that now routinely include millions of time steps and can measure Terabytes in size. These trajectories need to be analyzed using statistical mechanics approaches [6, 7] but because of the increasing size of data, trajectory analysis is becoming a bottleneck in typical biomolecular simulation scientific workflows [8]. Many data analysis tools and libraries have been developed to extract the desired information from the output trajectories from MD simulations [9–22] but few can efficiently use modern High Performance Computing (HPC) resources to accelerate the analysis stage. MD trajectory analysis primarily requires *reading* of data from the file system; the processed output data are typically negligible in size compared to the input data and therefore we exclusively investigate the reading aspects of trajectory I/O (i.e., the "I"). We focus on the *MDAnalysis* package [17, 18], which is an open-source object-oriented Python library for structural and temporal analysis of MD simulation trajectories and individual protein structures. Although *MDAnalysis* accelerates selected algorithms with OpenMP, it is not clear how to best use it for scaling up analysis on multi-node supercomputers. Here we discuss the challenges and lessons-learned for making parallel analysis on HPC resources feasible with *MDAnalysis*, which should also be broadly applicable to other general purpose trajectory analysis libraries.

Previously, we had used a parallel split-apply-combine approach to study the performance of the commonly performed "RMSD fitting" analysis problem [23, 24], which calculates the minimal root mean squared distance (RMSD) of the positions of a subset of atoms to a reference conformation under optimization of rigid body translations and rotations [7, 25, 26]. We had investigated two parallel implementations, one using *Dask* [27] and one using the message passing interface (MPI) with *mpi4py* [28, 29]. For both *Dask* and MPI, we had previously only been able to obtain good strong scaling performance within a single node. Beyond a single node performance had dropped due to *straggler* tasks, a subset of tasks that had performed abnormally slower than the typical task execution times; the total execution time had become dominated by stragglers and overall performance had decreased. Stragglers are a well-known challenge to improving performance on HPC resources [30] but there has been little discussion of their impact in the biomolecular simulation community.

In the present study, we analyzed the MPI case in more detail to better understand the origin of stragglers with the goal to find parallelization approaches to speed up parallel post-processing of MD trajectories in the *MDAnalysis* library. We especially wanted to make efficient use of the resources provided by current supercomputers such as multiple nodes with hundreds of CPU cores and a Lustre parallel file system.

As in our previous study [23] we selected the commonly used RMSD algorithm implemented in *MDAnalysis* as a typical use case. We employed the single program multiple data (SPMD) paradigm to parallelize this algorithm on three different HPC resources (XSEDE's *SDSC Comet*, *LSU SuperMic*, and *PSC Bridges* [31]). With SPMD, each process executes essentially the same operations on different parts of the data. The three clusters differed in their architecture but all used Lustre as their parallel file system. We used Python (`https://www.python.org/`), a machine-independent, byte-code interpreted, object-oriented programming language, which is well-established in the biomolecular simulation community with good support for parallel programming for HPC [28, 32]. We found that communication and reading I/O were the two main scalability bottlenecks, with some indication that read I/O might have been interfering with the communications. We therefore focused on two different approaches to mitigate I/O bottlenecks: MPI parallel I/O (MPI-IO) with the HDF5 file format and subfiling (trajectory file splitting). For subfiling, we obtained good results with the *Global Arrays* package [32, 33], which provides a convenient layer to access and manage arrays over multiple MPI ranks. Both MPI-IO and subfiling eliminated stragglers and improved the performance with near ideal scaling, $S(N) = N$, i.e., the speed-up S scaled linearly with the number N of CPU cores while exhibiting a slope of one.

The paper is organized as follows: We first review stragglers and existing approaches to parallelizing MD trajectory analysis in section 2. We describe the software packages and algorithms in section 3 and the benchmarking environment in section 4. Section 5 explains how we measured performance. The main results are presented in section 6, with section 7 demonstrating reproducibility on different supercomputers. We provide general guidelines and lessons-learned in section 8 and finish with conclusions in section 9.

2. Background and Related Work

In our previous work, we found that straightforward implementation of simple parallelization with a split-apply-combine algorithm in Python failed to scale beyond a single compute node [23] because a few tasks (MPI-ranks or Dask [27] processes) took much longer than the typical task and so limited the overall performance. However, the cause for these *straggler* tasks remained obscure. Here, we studied the straggler problem in the context of an MPI-parallelized trajectory analysis algorithm in Python and investigated solutions to overcome it. We briefly review stragglers in section 2.1 and summarize existing approaches to parallel trajectory analysis in section 2.2.

2.1. Stragglers

Stragglers or *outliers* were traditionally considered in the context of MapReduce jobs that consist of multiple tasks that all have to finish for the job to succeed: A straggler was a task that took an “unusually long time to complete” [34] and therefore substantially impeded job completion. In general, any component of a parallel workflow whose runtime exceeds a typical run time (for example, 1.5 times the median runtime) can be considered a straggler [35]. Stragglers are a challenge for improving performance on HPC resources [30]; they are a known problem in frameworks such as MapReduce [34, 35], Spark [36–39], Hadoop [34], cloud data centers [30, 40], and have a high impact on performance and energy consumption of big data systems [41]. Both internal and external factors are known to contribute to stragglers. Internal factors include heterogeneous capacity of worker nodes and resource competition due to other tasks running on the same worker node. External factors include resource competition due to co-hosted applications, input data skew, remote input or output source being too slow, faulty hardware [34, 42], and node mis-configuration [34]. Competition over scarce resources [35], in particular the network bandwidth, was found to lead to stragglers in writing on Lustre file systems [43]. Garbage collection [36, 37], Java virtual machine (JVM) positioning to cores [36], delays introduced while the tasks move from the scheduler to execution [38], disk I/O during shuffling, Java's just-in-time compilation [37], output skew [37], high CPU utilization, disk utilization, unhandled I/O access requests, and network package loss [30] were also among other external factors that might introduce stragglers. A wide variety of approaches have been investigated for detecting and mitigating stragglers, including tuning resource allocation and parallelism such as breaking the workload into many small tasks that are dynamically scheduled at runtime [44], slow Node-Threshold [34], speculative execution [34] and cause/resource-aware task management [35], sampling or data distribution estimation techniques, SkewTune to avoid data imbalance [45], dynamic work rebalancing [40], blocked time analysis [46], and intelligent scheduling [47].

In the present study, we analyzed large MD trajectories in parallel with MPI and Python and observed large variations in the completion time of individual MPI ranks. These variations bore some similarity to the straggler tasks observed in MapReduce frameworks so we approached analyzing and eliminating them in a similar fashion by systematically looking at different components of the problem, including read I/O from the shared Lustre file system and MPI communication. Even though we specifically worked in with the *MDAnalysis* package, all these principles and techniques are potentially applicable to MPI-parallelized data analysis in other Python-based libraries.

2.2. Other Packages with Parallel Analysis Capabilities

Different approaches to parallelizing the analysis of MD trajectories have been proposed. HiMach [14] introduces scalable and flexible parallel Python framework to deal with massive MD trajectories, by combining and extending Google's MapReduce and the VMD analysis tool [11]. HiMach's runtime is responsible to parallelize and distribute Map and Reduce classes to assigned cores. HiMach uses parallel I/O for file access during map tasks and MPI_Allgather in the reduction process. HiMach, however, does not discuss parallel analysis of analysis types that cannot be implemented via MapReduce. Furthermore, HiMach is not available under an open source license, which makes it difficult to integrate community contributions and add new state-of-the-art methods.

Wu et. al. [48] present a scalable parallel framework for distributed-memory post-simulation data analysis. This work consists of an interface that allows a user to write analysis programs sequentially, and the machinery that ensures these programs execute in parallel automatically. The main components of the proposed framework are (1) domain decomposition that splits computational domain into blocks with specified boundary conditions, (2) HDF5 based parallel I/O (3) data exchange that communicates ghost atoms between neighbor blocks, and (4) parallel analysis implementation of a real-world analysis application. This work does not discuss analysis methods which cannot be implemented using MapReduce and is limited to HDF5 file format.

Zazen [49] is a novel task-assignment protocol to overcome the I/O bottleneck for many I/O bound tasks. This protocol caches a copy of simulation output files on the local disks of the compute nodes of a cluster, and uses co-located data access with computation. Zazen is implemented in a parallel disk cache system and avoids the overhead associated with querying metadata servers by reading data in parallel from local disks. This approach has also been used to improve the performance of HiMach [14]. It, however, advocates a specific architecture where a parallel supercomputer, which runs the simulations, immediately pushes the trajectory data to a local analysis cluster where trajectory fragments are cached on node-local disks. In the absence of such a specific workflow, one would need to stage the trajectory across nodes, and the time for data distribution is likely to reduce any gains from the parallel analysis.

VMD [11, 50] provides molecular visualization and analysis tool through algorithmic and memory efficiency improvements, vectorization of key CPU algorithms, GPU analysis and visualization algorithms, and good parallel I/O performance on supercomputers. It is one of the most advanced programs for the visualization and analysis of MD simulations. It is, however, a large monolithic program, that can only be driven through its built-in Tcl interface and thus is less well suited as a library that allows the rapid development of new algorithms or integration into workflows.

CPPTraj [19] offers multiple levels of parallelization (MPI and OpenMP) in a monolithic C++ implementation. CCPTraj allows parallel reads between frames of the same trajectory but is especially geared towards processing an ensemble of many trajectories in parallel.

pyPcazip [51] is a suite of software tools written in Python for compression and analysis of MD simulation data, in particular ensembles of trajectories. pyPcazip is MPI parallelised and is specific to PCA-based investigations of MD trajectories and supports a wide variety of trajectory file formats (based on the capabilities of the underlying mdtraj package [20]). pyPcazip can take one or many input MD trajectory files and convert them into a highly compressed, HDF5-based pcz format with insignificant loss of information. However, the package does not support general purpose analysis.

In situ analysis is an approach to execute analysis simultaneously with the running MD simulation so that I/O bottlenecks are mitigated [52, 53]. Malakar *et al.* studied the scalability challenges of time and space shared modes of analyzing large-scale MD simulations through a topology-aware mapping for simulation and analysis using the LAMMPS code [52]. Similarly, Taufer and colleagues [53] presented their own framework for *in situ* analysis, which is based on the fast on-the-fly calculation of metadata that characterizes protein substructures via maximum eigenvalues of distance matrices. These metadata are used to index trajectory frames and enable targeted analysis of trajectory subsets. Both studies provide important ideas and approaches towards moving towards online-analysis in conjunction with a running simulation but are limited in generality.

All of the above frameworks provide tools for parallel analysis of MD trajectories. These frameworks, however, tend to fall short in providing parallelism in the context of a general and flexible library for the analysis of MD trajectories. Although straggler tasks are a common challenge arising in parallel analysis and are well-known in the data analysis community (see Section 2.1), there is, to our knowledge, little discussion about this problem in the biomolecular simulation community. Our own experience with a MapReduce approach in *MDAnalysis* [23] suggested that stragglers might be a somewhat under-appreciated problem. Therefore, in the present work we want to better understand requirements for efficient parallel analysis of MD trajectories in *MDAnalysis*, but to also provide more general guidance that could benefit developments in other libraries inside and outside of the scope of analysis of MD simulations.

3. Algorithms and Software Packages

For our investigation of parallel trajectory analysis we focus on using MPI as the standard approach to parallelization in HPC. We employ the Python language, which is widely used in the scientific community because it facilitates rapid development of small scripts and code prototypes as well as development of large applications and highly portable and reusable modules and libraries. We use the *MDAnalysis* library to calculate a "RMSD timeseries" (explained in section 3.1) as a representative use case. Further details on the software packages are provided in sections 3.2–3.4.

3.1. RMSD Calculation with MDAnalysis

Simulation data exist in trajectories in the form of time series of atom positions and sometimes velocities. Trajectories come in a plethora of different and idiosyncratic file formats. *MDAnalysis* [17, 18] is a widely used open source library to analyze trajectory files with an object oriented interface. The library is written in Python, with time critical code in C/C++/Cython. *MDAnalysis* supports most file formats of simulation packages including CHARMM [54], Gromacs [55], Amber [56], and NAMD [57] and the Protein Data Bank [58] format. At its core, it reads trajectory data in different formats and makes them available through a uniform API; specifically, coordinates are represented as standard NumPy arrays [59].

As a test case that is representative of a common task in the analysis of biomolecular simulation trajectories we calculated the timeseries of the minimal structural root mean

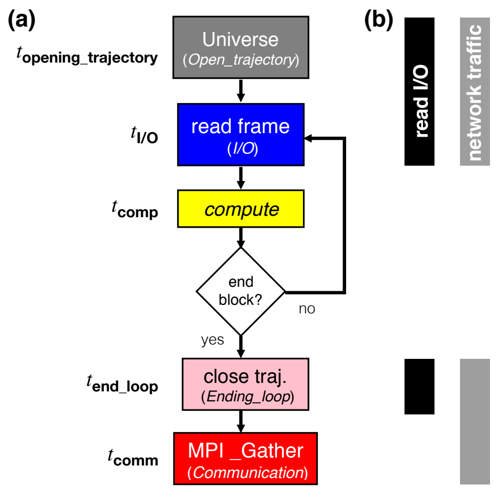


Figure 1: Flow chart of the MPI-parallelized RMSD algorithm, Algorithm 1. **(a)** Each MPI process performs the same steps but reads trajectory frames from different blocks of the trajectory. The color scheme and labels in italics correspond to the colors and labels for measured timing quantitities in the following graphs (e.g., Figs. 2c and 2d). The names of the corresponding timing quantitities from Table 3 are listed next to each step. **(b)** Steps that access the shared Lustre file system with read I/O are included in the black bars; steps that communicate via the shared InfiniBand network are included in the gray bars. The Lustre file system is accessed through the network and hence all I/O steps also use the network.

square distance (**RMSD**) after rigid body superposition [7, 26]. The RMSD is used to show the rigidity of protein domains and more generally characterizes structural changes. It is calculated as a function of time t as

$$\mathrm{RMSD}(t) = \min_{\mathsf{R},\mathbf{t}} \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left[(\mathsf{R}\cdot\mathbf{x}_i(t)+\mathbf{t})-\mathbf{x}_i^{\mathrm{ref}}\right]^2} \qquad (1)$$

where $\mathbf{x}_i(t)$ is the position of atom i at time t, $\mathbf{x}_i^{\mathrm{ref}}$ is its position in a reference structure and the distance between these two is minimized by finding the optimum 3×3 rotation matrix R and translation vector $\mathbf{t}$. The optimum rigid body superposition was calculated with the QCPROT algorithm [25, 60] (implemented in Cython and available through the `MDAnalysis.analysis.rms` module [18]).

The RMSD trajectory analysis was parallelized as outlined in the flow chart in Figure 1, with further details available in Algorithm 1. Each MPI process loads the core MDAnalysis structure (called the `Universe`), which includes loading a shared "topology" file with the simulation system information and opening the shared trajectory file. Each process operates on a different block of frames and iterates through them by reading the coordinates of a single frame into memory and performing the RMSD computation with them. Once all frames in the block are processed, the trajectory file is closed and results are communicated to MPI rank 0 using `MPI_Gather()`.

The RMSD was determined for a subset of protein atoms, the $N = 214$ C_α atoms of our test system (see section 4.3 for details). The time complexity for the RMSD Algorithm 1 is $O(T \times N^2)$ [25] where T is the number of frames in the trajectory and N the number of particles included in the RMSD calculation.

Algorithm 1 MPI-parallel Multi-frame RMSD Algorithm

```
    Input: size: Total number of frames
    ref: mobile group in the initial frame which will be considered as reference
    start & stop: Starting and stopping frame index
    topology & trajectory: files to read the data structure from
    Output: Calculated RMSD arrays
 1: procedure Block_RMSD(topology, trajectory, ref, start, stop)
 2:     u ← Universe(topology, trajectory)        ▷ u hold all the information of the physical system
 3:     g ← u.frames[start:stop]
 4:     for ∀iframe in g do
 5:         results[iframe] ← RMSD(g, ref)
 6:     end for
 7:     return results
 8: end procedure
 9:
10: MPI Init
11: rank ← rank ID
12: index ← indices of mobile atom group
13: xref0 ← Reference atom group's position
14: out ← Block_RMSD(topology, trajectory, xref0, start=start, stop=stop)
15:
16: Gather(out, RMSD_data, rank_ID=0)
17: MPI Finalize
```

3.2. MPI for Python (mpi4py)

MPI for Python (*mpi4py*) is a Python wrapper for the Message Passing Interface (MPI) standard and allows any Python program to employ multiple processors [28, 29]. Performance degradation due to using *mpi4py* is not prohibitive [28, 29] and the overhead is far smaller than the overhead associated with the use of interpreted versus compiled languages [32]. Overheads in *mpi4py* are small compared to C code if efficient raw memory buffers are used for communication [28], as used in the present study.

3.3. Global Arrays Toolkit

The *Global Arrays* (GA) toolkit provides users with a language interface that allows them to distribute data while maintaining the type of global index space and programming syntax similar to what is available when programming on a single processor [33]. *Global Arrays* is implemented with Fortran-77 and C bindings and provides C++ and Python interfaces. It allows manipulating physically distributed dense multi-dimensional arrays without explicitly defining communication and synchronization between processes. The underlying communication is determined by a runtime environment, which defaults to the *Communication runtime for Extreme Scale* (ComEx) [61]. ComEx uses shared memory for intra-node communication and inter-node communication employs ComEx with MPI. *Global Arrays in NumPy* (GAiN) extends GA to Python through Numpy [32]. The *Global Arrays* toolkit provides functions to create global arrays (`ga_create()`) and to copy data to (`ga_put()`) and from (`ga_get()`) such a global array, as well as additional functions for copying between arrays and freeing them [32]. When a global array is created (`ga_create()`) each process will create an array of the same shape and size, physically located in the local memory space of that process [33]. The GA library maintains a list of all these memory locations, which can be queried with the `ga_access()` function. Using a pointer returned by `ga_access()`, one can directly modify the data that is local to each process. When a process tries to access a block of data the request is first decomposed into individual blocks representing the contribution to the total request from the data held locally on each process (*B. J. Palmer and J. Daily, personal communication*). The requesting process then makes individual requests to each of the remote processes.

GA allows independent, asynchronous, and efficient access to logical blocks of physically distributed arrays, with no need for explicit cooperation by other processes; in particular, it allows data locality to be explicitly specified and used [62]. We investigated if communication cost could be reduced by using *Global Arrays.* Algorithm 2 describes the RMSD algorithm with *Global Arrays* instead of MPI.

Algorithm 2 MPI-parallel Multi-frame RMSD using Global Arrays

```
Input:size: Total number of frames assigned to each rank $N_b$
    g_a: Initialized Global Arrays
    xref0: mobile group in the initial frame which will be considered as reference
    start & stop: that tell which block of trajectory (frames) is assigned to each rank
    topology & trajectory: files to read the data structure from
Include: Block_RMSD() from Algorithm 1
1: bsize ← ceil(trajectory.number_frames / size)
2: g_a ← ga.create(ga.C_DBL, [bsize*size,2], "RMSD")
3: buf ← np.zeros([bsize*size,2], dtype=float)
4: out ← Block_RMSD(topology, trajectory, xref0, start=start, stop=stop)
5: ga.put(g_a, out, (start,0), (stop,2))
6: if rank == 0 then
7:     buf ← ga.get(g_a, lo=None, hi=None)
8: end if
```

3.4. MPI and Parallel HDF5

HDF5 is a structured self-describing hierarchical data format which is the standard mechanism for storing large quantities of numerical data in Python (`http://www.hdfgroup.org/HDF5`, [63]). Parallel HDF5 (*PHDF5*) typically sits on top of a MPI-IO layer and can use MPI-IO optimizations. In *PHDF5*, all file access is coordinated by the MPI library; otherwise, multiple processes would compete over accessing the same file on disk. MPI-based applications launch multiple parallel instances of the Python interpreter that communicate with each other via the MPI library. Implementation is straightforward as long as the user supplies a MPI communicator and takes into account some constraints required for data consistency [63]. *HDF5* itself handles nearly all the details involved with coordinating file access when the shared file is opened through the *mpio* driver.

MPI has two flavors of operation: collective (all processes have to participate in the same order) and independent (processes can perform the operation in any order or not at all) [63]. With *PHDF5*, modifications to file metadata must be performed collectively and although all processes perform the same task, they do not need to be synchronized [63]. Other tasks and any type of data operations can be performed independently by processes. In the present study, we use independent operations.

4. Benchmark Environment

Our benchmark environment consisted of three different XSEDE [31] HPC resources (described in section 4.1), the software stack used (section 4.2), which had to be compiled for each resource, and the common test data set (section 4.3).

4.1. HPC Resources

The computational experiments were executed on standard compute nodes of three XSEDE [31] supercomputers, *SDSC Comet*, *PSC Bridges*, and *LSU SuperMIC* (Table 1). *SDSC Comet* is a 2 PFlop/s cluster with 2,020 compute nodes in total. It is optimized for running a large number of medium-size calculations (up to 1,024 cores) to support the most prevalent type of calculation on XSEDE resources. *PSC Bridges* is a 1.35 PFlop/s cluster with different types of computational nodes, including 16 GPU nodes, 8 large memory and 2 extreme memory nodes, and 752 regular nodes. It was designed to flexibly support both traditional (medium scale calculations) and non-traditional (data analytics) HPC uses. *LSU SuperMIC* offers 360 standard compute nodes with a peak performance of 557 TFlop/s. The parallel file system on all three machines is Lustre (`http://lustre.org/`) and is shared between the nodes of each cluster.

Name	Nodes	Number of Nodes	CPUs	RAM	Network Topology	Scheduler and Resource Manager	parallel file system
SDSC Comet	Compute	6400	2 Intel Xeon (E5-2680v3) 12 cores/CPU, 2.5 GHz	128 GB DDR4 DRAM	56 Gbps IB	SLURM	Lustre
PSC Bridges	RSM	752	2 Intel Haswell (E5-2695 v3) 14 cores/CPU, 2.3 GHz	128 GB, DDR4-2133Mhz	12.37 Gbps OPA	SLURM	Lustre
LSU SuperMIC	Standard	360	2 Intel Ivy Bridge (E5-2680) 10 cores/CPU, 2.8 GHz	64 GB, DDR3-1866Mhz	56 Gbps IB	PBS	Lustre

Table 1: Configuration of the HPC resources that were benchmarked. Only a subset of the total available nodes were used. IB: InfiniBand; OPA: Omni-Path Architecture.

4.2. Software

Table 2 lists the tools and libraries that were required for our computational experiments. Many domain specific packages are not available in the standard software installation on supercomputers. We therefore had to compile them, which in some cases required substantial effort due to non-standard building and installation procedures or lack of good documentation. Because this is a common problem that hinders reproducibility we provide detailed version information, notes on the installation process, as well as comments on the ease of installation and the quality of the documentation in Table 2. For the MPI implementation we used Open MPI release 1.10.7 (`https://www.open-mpi.org/`) consistently everywhere. Detailed instructions to create the computing environments together with the benchmarking code can be found in the GitHub repository. Carefully setting up the same software stack on the three different supercomputers allowed us to clearly demonstrate the reproducibility of our results and showed that our findings were not dependent on machine specifics.

Package	Version	Description	Ease of Installation	Documentation	Installation	Dependencies
GCC	4.9.4	GNU Compiler Collection	0	++	via configuration files, environment or command line options, minimal configuration is required	–
Open MPI	1.10.7	MPI Implementation	0	++	via configuration files, environment or command line options, minimal configuration is required	–
Global Arrays	5.6.1	Global Arrays	–	+	via configuration files, environment or command line options, several optional configuration settings available	MAMA, ARMCI MPI 1.x/2.x/3.x implementation like Open MPI built with shared/dynamic libraries, GCC
Python	2.7.13	Python language	+	++	Conda Installation	–
MPI4py	3.0.0	MPI for Python	+	++	Conda Installation	Python 2.7 or above, MPI 1.x/2.x/3.x implementation like Open MPI built with shared/dynamic libraries, Cython
GA4py	1.0	Global Arrays for Python	0	0	Python Setuptools	Global Arrays, Python 2 only, MPI 1.x/2.x/3.x implementation like Open MPI built with shared/dynamic libraries, Cython, MPI4py, Numpy
PHDF5	1.10.1	Parallel HDF5	–	++	via configuration files, environment or command line options, several optional configuration settings available	MPI 1.x/2.x/3.x implementation like Open MPI GNU, MPIF90, MPICC, MPICXX
H5py	2.7.1	Pythonic wrapper around the HDF5	+	++	Conda Installation	Python 2.7, or above, PHDF5, Cython
MDAnalysis	0.17.0	Python library to analyze trajectories from MD simulations	+	++	Conda Installation	Python >=2.7, Cython, GNU, Numpy

Table 2: Detailed comparison on the dependencies and installation of different software packages used in the present study. Software was built from source or obtained via a package manager and installed on the multi-user HPC systems in Table 1. Evaluation of ease of installation and documentation uses a subjective scale with "++" (excellent), "+" (good), "0" (average), and "−" (difficult/lacking) and reflects the experience of a typical domain scientist at the graduate/post-graduate level in a discipline such as computational biophysics or chemistry.

4.3. Data Set

The test system contained the protein adenylate kinase with 214 amino acid residues and 3341 atoms in total [64] and the topology information (atoms types and bonds) was stored in a file in CHARMM PSF format. The test trajectory was created by concatenating 600 copies of a MD trajectory with 4,187 time frames (saved every 240 ps for a total simulated time of 1.004 μs) in CHARMM DCD format [65] and converting to Gromacs XTC format trajectory, as described for the "600x" trajectory in Khoshlessan et al. [23]. The trajectory had a file size of about 30 GB and contained 2,512,200 frames (corresponding to 602.4 μs simulated time). The file size was relatively small because water molecules that were also part of the original MD simulations were stripped to reduce the original file size by a factor of about 10; such preprocessing is a common approach if one is only interested in the protein behavior. Thus, the trajectory represents a small to medium system size in the number of atoms and coordinates that have to be loaded into memory for each time frame. The XTC format is a format with lossy compression [66, 67], which also contributed to the compact file size. XTC trades lower I/O demands for higher CPU demands during decompression and therefore performed well in our previous study [23]. Although 2,512,200 frames represents a long simulation for current standards, such trajectories will become increasingly common due to the use of special hardware [68, 69] and GPU-acceleration [55, 70, 71].

5. Methods

Documentation and benchmark codes are made available in the code repository `https://github.com/hpcanalytics/supplement-hpc-py-parallel-mdanalysis` under the GNU General Public License v3.0 (code) and the Creative Commons Attribution-ShareAlike (documentation). These materials should enable users to recreate the computational environment on the tested XSEDE HPC resources (*SDSC Comet*, *PSC Bridges*, *LSU SuperMIC*), prepare data files, and run the computational experiments.

In the following we define the quantities and approach used for our performance measurements, with a full summary of all definitions in Table 3. We evaluated MPI performance of the parallel RMSD timeseries algorithm 1 by timing the total time to solution as well as the execution time for different parts of the code for individual MPI ranks with the help of the Python `time.time()` function.

5.1. Timing Observables

We abbreviate the timings in the following as variables $t_{\mathrm{L}n}$ where Ln refers to the line number in algorithm 1. We measured in the function `block_rmsd()` the *read I/O*

Quantity	Definition
N_b	$N_\mathrm{frames}^\mathrm{total}/N$
$t_\mathrm{end\_loop}$	t_L6
$t_\mathrm{opening\_trajectory}$	$t_\mathrm{L2} + t_\mathrm{L3}$
t_comp	$\sum_{\mathrm{frame}=1}^{N_\mathrm{b}} t_\mathrm{comp}^\mathrm{frame}$
$t_\mathrm{I/O}$	$\sum_{\mathrm{frame}=1}^{N_\mathrm{b}} t_\mathrm{I/O}^\mathrm{frame}$
$t_\mathrm{all\_frame}$	$t_\mathrm{L4} + t_\mathrm{L5} + t_\mathrm{L6}$
t_RMSD	$t_\mathrm{L1} + ... + t_\mathrm{L8}$
$t_\mathrm{comm/MPI}$	t_L16
$t_\mathrm{comm/GA}$	$t_\mathrm{L5} + t_\mathrm{L6} + t_\mathrm{L7} + t_\mathrm{L8}$
t_comm	$t_\mathrm{comm/MPI}$ (Alg. 1) or $t_\mathrm{comm/GA}$ (Alg. 2)
$t_\mathrm{Overhead1}$	$t_\mathrm{all\_frame} - t_\mathrm{I/O} - t_\mathrm{comp} - t_\mathrm{end\_loop}$
$t_\mathrm{Overhead2}$	$t_\mathrm{RMSD} - t_\mathrm{all\_frame} - t_\mathrm{opening\_trajectory}$
t_N	$t_\mathrm{RMSD} + t_\mathrm{comm}$
$\overline{t_\mathrm{comp}}$	$\frac{1}{N}\sum_{\mathrm{rank}=1}^{N} t_\mathrm{comp}$
$\overline{t_\mathrm{I/O}}$	$\frac{1}{N}\sum_{\mathrm{rank}=1}^{N} t_\mathrm{I/O}$
$\overline{t_\mathrm{comm}}$	$\frac{1}{N}\sum_{\mathrm{rank}=1}^{N} t_\mathrm{comm}$
t_total	$\max t_N$

Table 3: Summary of measured timing quantitities. Timings are collected for the specified line numbers in the code, labelled as $t_{\mathrm{L}n}$ where Ln refers to the line number in the corresponding algorithm. $t_\mathrm{comm/MPI}$ (in Algorithm 1) and $t_\mathrm{comm/GA}$ (in Algorithm 2) are both referred to as t_comm in the text. Variables in the top half of the table refer to measurements of an individual MPI rank. Variables in the bottom half are aggregates such as averages over all ranks or the total time to solution.

time for ingesting the data of one trajectory frame from the file system into memory, $t_\mathrm{I/O}^\mathrm{frame} = t_\mathrm{L4}$, and the *compute time* per trajectory frame to perform the computation, $t_\mathrm{comp}^\mathrm{frame} = t_\mathrm{L5}$. The *total read I/O time for a MPI rank*, $t_\mathrm{I/O} = \sum_{\mathrm{frame}=1}^{N_\mathrm{b}} t_\mathrm{I/O}^\mathrm{frame}$, is the sum over all I/O times for all the N_frames frames assigned to the rank; similarly, the *total compute time for a MPI rank* is $t_\mathrm{comp} = \sum_{\mathrm{frame}=1}^{N_\mathrm{b}} t_\mathrm{comp}^\mathrm{frame}$. The time delay between the end of the last iteration and exiting the `for` loop is $t_\mathrm{end\_loop} = t_\mathrm{L6}$. The time $t_\mathrm{opening\_trajectory} = t_\mathrm{L2} + t_\mathrm{L3}$ measures the problem setup, which includes data structure initialization and opening of topology and trajectory files. The *communication time*, $t_\mathrm{comm} = t_\mathrm{L16}$, is the time to gather all data from all processor ranks to rank zero. The total time (for all frames) spent in `block_rmsd()` is $t_\mathrm{RMSD} = \sum_{i=1}^{8} t_{\mathrm{L}i}$. There are parts of the code in `block_rmsd()` that are not covered by the detailed timing information of t_comp and $t_\mathrm{I/O}$. Unaccounted time is considered as *overhead*. We define $t_\mathrm{Overhead1}$ and $t_\mathrm{Overhead2}$ as the overheads of the calculations (see Table 3 for the definitions); both are expected to be negligible, which was the case in all our measurements. Finally, the *total time* *to completion of a single MPI rank*, when utilizing N cores for the execution of the overall experiment, is t_N, and as a result $t_{\text{RMSD}} + t_{\text{comm}} \equiv t_N$.

5.2. Performance Parameters

We measured the *total time to solution* $t_{\text{total}}(N)$ with N MPI processes on N cores, which is effectively $t_{\text{total}}(N) \approx \max(t_N)$. Strong scaling was quantified by the speed-up

$$S(N) = \frac{t_{\text{total}}(1)}{t_{\text{total}}(N)}, \tag{2}$$

relative to performance on a single core ($t_{\text{total}}(1)$), and the efficiency

$$E(N) = \frac{S(N)}{N}. \tag{3}$$

Averages over ranks were calculated as

$$\overline{t_{\text{comp}}} = \frac{1}{N} \sum_{\text{rank}=1}^{N} t_{\text{comp}} = \frac{1}{N} \sum_{\text{rank}=1}^{N} \sum_{\text{frame}=1}^{N_{\text{b}}} t_{\text{comp}}^{\text{frame}}, \tag{4}$$

$$\overline{t_{\text{I/O}}} = \frac{1}{N} \sum_{\text{rank}=1}^{N} t_{\text{I/O}} = \frac{1}{N} \sum_{\text{rank}=1}^{N} \sum_{\text{frame}=1}^{N_{\text{b}}} t_{\text{I/O}}^{\text{frame}}, \tag{5}$$

and

$$\overline{t_{\text{comm}}} = \frac{1}{N} \sum_{\text{rank}=1}^{N} t_{\text{comm}}. \tag{6}$$

Additionally, we introduced two performance parameters that we found to be indicative of the occurrence of stragglers. We defined the ratio of compute time to read I/O time for the serial code as

$$R_{\text{comp/IO}} = \frac{t_{\text{comp}}}{t_{\text{I/O}}} = \frac{t_{\text{comp}}/N_{\text{frames}}^{\text{total}}}{t_{\text{I/O}}/N_{\text{frames}}^{\text{total}}} = \frac{\overline{t_{\text{comp}}^{\text{frame}}}}{\overline{t_{\text{I/O}}^{\text{frame}}}} \tag{7}$$

where the last equality shows that the ratio can also be computed from the average times per frame, $\overline{t_{\text{comp}}^{\text{frame}}}$ and $\overline{t_{\text{I/O}}^{\text{frame}}}$. $R_{\text{comp/IO}}$ was calculated with the serial versions of our algorithms (on a single CPU core) in order to characterize the computational problem in the absence of parallelization. The ratio of compute to communication time was defined by the ratio of average total compute time to the average total communication time

$$R_{\text{comp/comm}} = \frac{\overline{t_{\text{comp}}}}{\overline{t_{\text{comm}}}}. \tag{8}$$

Because t_{comm} cannot be measured for a serial code, we estimated $R_{\text{comp/comm}}$ from the rank-averages (Eqs. 4 and 6) for a given number of MPI ranks.

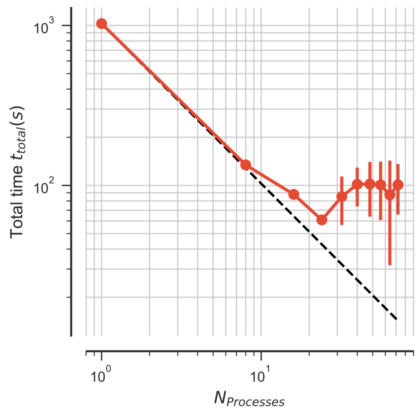
6. Computational Experiments

We had previously measured the performance of the MPI-parallelized RMSD analysis task on two different HPC resources (*SDSC Comet* and *TACC Stampede*) and had found that it only scaled well up to a single node due to high variance in the runtime of the MPI ranks, similar to the straggler phenomenon observed in big-data analytics [23]. However, the ultimate cause for this high variance could not be ascertained. We therefore performed more measurements with more detailed timing information (see section 5) on *SDSC Comet* (described in this section) and two other supercomputers (summarized in section 7) in order to better understand the origin of the stragglers and find solutions to overcome them.

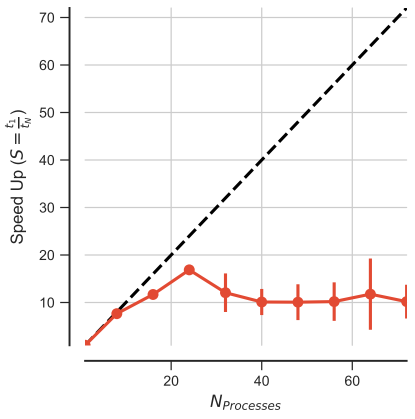
6.1. RMSD Benchmark

We measured strong scaling for the RMSD analysis task (Algorithm 1) with the 2,512,200 frame test trajectory (section 4.3) on 1 to 72 cores (one to three nodes) of *SDSC Comet* (Figures 2a and 2b). We observed poor strong scaling performance beyond a single node (24 cores), comparable to our previous results [23]. A more detailed analysis showed that the RMSD computation, and to a lesser degree the read I/O, considered on their own, scaled well beyond 50 cores (yellow and blue lines in Figure 2c). But communication (sending results back to MPI rank 0 with `MPI_Gather()`; red line in Figure 2c) and the initial file opening (loading the system information into the `MDAnalysis.Universe` data structure from a shared "topology" file and opening the shared trajectory file; gray line in Figure 2c) started to dominate beyond 50 cores.

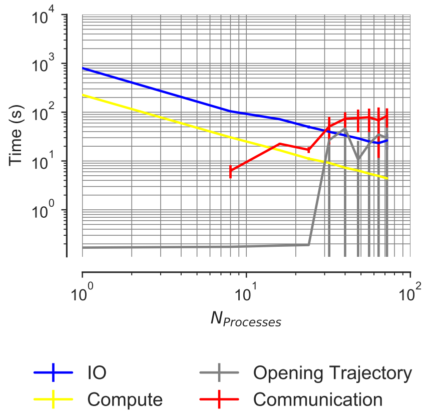
Communication cost and initial time for opening the trajectory were distributed unevenly across MPI ranks, as shown in Figure 2d. The ranks that took much longer to complete than the typical execution time of the other ranks were the stragglers that hurt performance.



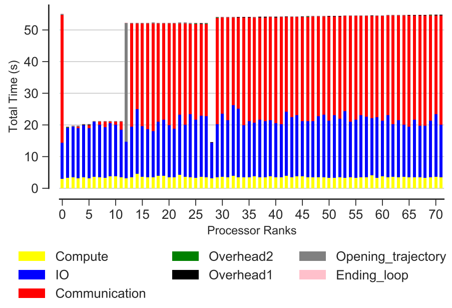
(a) Scaling total (five repeats)



(b) Speed-up (five repeats)



(c) Scaling for different components (five repeats)



(d) Time comparison on different parts of the calculations per MPI rank (example)

Figure 2: Performance of the RMSD task parallelized with MPI on *SDSC Comet*. Results were communicated back to rank 0. Five independent repeats were performed to collect statistics. (a-c) The error bars show standard deviation with respect to the mean. In serial, there is no communication and no data points are shown for $N = 1$ in (c). (d) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for definitions. These are data from one run of the five repeats. MPI ranks 0, 12–27 and 29–72 are stragglers.

We qualitatively denoted by *straggler* any MPI rank that took at least about twice as long as the group of ranks that finished fastest, roughly following the original description of a straggler as a task that took an “unusually long time to complete” [34]. The fast-finishing ranks were generally clearly distinguishable in the per-rank timings such as in Figures 2d and A.11d. Such a qualitative definition of stragglers was sufficient for our purpose to identify scalability bottlenecks, as shown in the following discussion.

Identification of Scalability Bottlenecks

In the example shown in Figure 2d, 62 ranks out of 72 took about 60 s (the stragglers) whereas the remaining ranks only took about 20 s. In other instances, far fewer ranks were stragglers, as shown, for example, in Figure A.11d. The detailed breakdown of the time spent on each rank (Figure 2d) showed that the computation, t_{comp}, was relatively constant across ranks. The time spent on reading data from the shared trajectory file on the Lustre file system into memory, $t_{\text{I/O}}$, showed variability across different ranks. The stragglers, however, appeared to be defined by occasionally much larger *communication* times, t_{comm} (line 16 in Algorithm 1), which were on the order of 30 s, and by larger times to initially open the trajectory (line 2 in Algorithm 1). t_{comm} varied across different ranks and was barely measurable for a few of them. Although the data in Figure 2d represented one run and in other instances different number of ranks were stragglers, the averages over all ranks in five independent repeats (Figure 2c) showed that increased t_{comm} were generally the reason for large variations in the run time for each rank. This initial analysis indicated that communication was a major issue that prevented good scaling beyond a single node but the problems related to file I/O also played an important role in limiting scaling performance.

Influence of Hardware

We ran the same benchmarks on multiple HPC systems that were equipped with a Lustre parallel file system [XSEDE’s *PSC Bridges* (Fig. A.11) and *LSU SuperMIC* (Fig. A.12)], and observed the occurrence of stragglers, in a manner very similar to the results described for *SDSC Comet*. There was no clear pattern in which certain MPI ranks would always be a straggler, and neither could we trace stragglers to specific cores or nodes. Therefore, the phenomenon of stragglers in the RMSD case was reproducible on different clusters and thus appeared to be independent from the underlying hardware.

6.2. *Effect of Compute to I/O Ratio on Performance*

The results in section 6.1 indicated opening the trajectory, communication, and read I/O to be important factors that appeared to correlate with stragglers. In order to better characterize the RMSD task, we computed the ratio between the time to complete the computation and the time spent on I/O per frame. The average values were $\overline{t^{\text{frame}}_{\text{comp}}} = 0.09$ ms, $\overline{t^{\text{frame}}_{\text{IO}}} = 0.3$ ms, resulting in a compute-to-I/O ratio $R_{\text{comp/IO}} \approx 0.3$ (Eq. 7). Because $R_{\text{comp/IO}} \ll 1$, the RMSD analysis task was characterized as I/O bound.

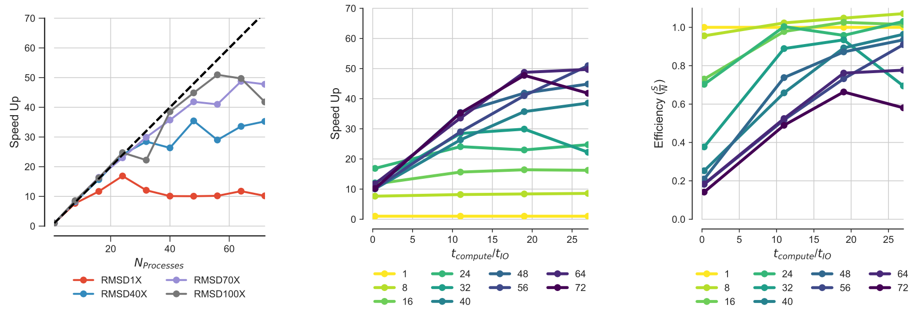
As we were not able to achieve good scaling beyond a single node, we hypothesized that decreasing the I/O load relative to the compute load would interleave read I/O with longer periods of computation, thus reducing the impact of I/O contention and the impact of stragglers. We therefore set out to measure compute bound tasks, i.e. ones with $R_{\text{comp/IO}} \gg 1$. To measure the effect of the $R_{\text{comp/IO}}$ ratio on performance but leaving other parameters the same, we artificially increased the computational load by repeating the same RMSD calculation (line 10, algorithm 1) 40, 70 and 100 times in a loop, resulting in forty-fold ("40×"), seventy-fold ("70×"), and one hundred-fold ("100×") load increases.

6.2.1. *Effect of Increased Compute Workload*

For an X-fold increase in workload, we expected the workload for the computation to scale with X as $t_{\text{comp}}(X) = N^{\text{total}}_{\text{frames}} X \overline{t^{\text{frame}}_{\text{comp}}}$ while the read I/O workload $t_{\text{I/O}}(X) = N^{\text{total}}_{\text{frames}} \overline{t^{\text{frame}}_{\text{I/O}}}$ (number of frames times the average time to read a frame) should remain independent of X. Therefore, the ratio for any X should be $R_{\text{comp/IO}}(X) = t_{\text{comp}}(X)/t_{\text{I/O}}(X) = X R_{\text{comp/IO}}(X = 1)$, i.e., $R_{\text{comp/IO}}$ should just linearly scale with the workload factor X. The measured $R_{\text{comp/IO}}$ ratios of 11, 19, 27 for the increased computational workloads agreed with this theoretical analysis, as shown in Table 4.

Workload X	t_{comp} **(s)**	$t_{\text{I/O}}$ **(s)**	$R_{\text{comp/IO}}$	
			measured	theoretical
1×	226	791	0.29	
40×	8655	791	11	11
70×	15148	791	19	20
100×	21639	791	27	29

Table 4: Change in $R_{\text{comp/IO}}$ ratio with change in the RMSD workload X. The RMSD workload was artificially increased in order to examine the effect of compute to I/O ratio on the performance. The reported compute and I/O time were measured based on the serial version using one core. The theoretical $R_{\text{comp/IO}}$ (see text) is provided for comparison.



(a) Speed-Up

(b) Speed-Up

(c) Efficiency

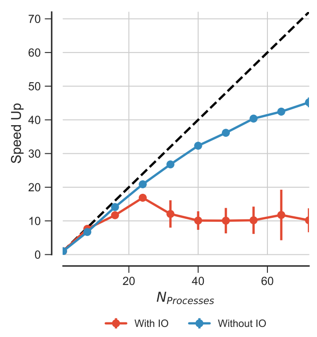
Figure 3: Effect of $R_{\text{comp/IO}}$ ratio on performance of the RMSD task on *SDSC Comet*. We tested performance for $R_{\text{comp/IO}}$ ratios of 0.3, 11, 19, 27, which correspond to 1× RMSD, 40× RMSD, 70× RMSD, and 100× RMSD respectively. (a) Effect of $R_{\text{comp/IO}}$ on the speed-up. (b) Change in speed-up with respect to $R_{\text{comp/IO}}$ for different processor counts. (c) Change in the efficiency with respect to $R_{\text{comp/IO}}$ for different processor counts.

We performed the experiments with increased workload to measure the effect of the $R_{\text{comp/IO}}$ ratio on performance (Figure 3). The strong scaling performance as measured by the speed-up $S(N)$ improved with increasing $R_{\text{comp/IO}}$ ratio (Figure 3a). The calculations consistently showed better scaling performance to larger numbers of cores for higher $R_{\text{comp/IO}}$ ratios, e.g., $N = 56$ cores for the 70× RMSD task. The speed-up and efficiency approached their ideal value for each processor count with increasing $R_{\text{comp/IO}}$ ratio (Figures 3b and 3c). Even for moderately compute-bound workloads, such as the 40× and 70× RMSD tasks, increasing the computational workload over I/O reduced the impact of stragglers even though they still contributed to large variations in timing across different ranks and thus irregular scaling.

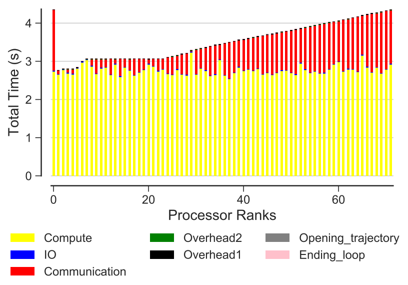
We also investigated the influence of the ratio of compute to communication costs

($R_{\text{comp/comm}}$, Eq. 8) on performance in Appendix B. We found evidence to support the hypothesis that a larger ratio was beneficial, provided I/O costs could also be reduced. However, read I/O ultimately seemed to be the key determinant for performance, as discussed in the next sections.

6.2.2. Effect of Absence of Read I/O on Communication



(a) Speed up comparison



(b) Time comparison on different parts of the calculations per MPI rank when I/O is removed

Figure 4: Comparison of the performance of the RMSD task with I/O ($R_{\text{comp/IO}} \approx 0.3$) and without I/O ($R_{\text{comp/IO}} = +\infty$) on *SDSC Comet*. Five repeats were performed to collect statistics. (a) Speed-up. The error bars show standard deviation with respect to the mean. (b) Compute t_{comp}, read I/O $t_{\text{I/O}} = 0$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank. (See Table 3 for definitions.)

In order to study an extreme case of a compute-bound task, we eliminated all I/O from the RMSD task by randomly generating artificial trajectory data in memory; the data had the the same size as if they had been obtained from the trajectory file. The time for the data generation was excluded and no file access was necessary. Without any I/O, performance improved markedly (Figure 4), with reasonable scaling up to 72 cores (3 nodes). No stragglers were observed because overall communication time decreased and showed less variability; nevertheless, an increase in communication time prevented ideal scaling performance. Although in practice I/O cannot be avoided, this experiment demonstrated that accessing the trajectory file on the Lustre file system is at least one cause for the observed stragglers.

6.3. Reducing I/O Cost

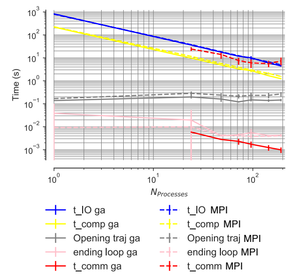
In order to improve performance we needed to employ strategies to avoid the competition over file access across different ranks when the $R_{\text{comp/IO}}$ ratio was small. To this end, we experimented with two different ways for reducing the I/O cost: 1) splitting the trajectory file into as many segments as the number of processes, thus using file-per-process access, and 2) using the HDF5 file format together with MPI-IO parallel reads instead of the XTC trajectory format. We discuss these two approaches and their performance improvements in detail in the following sections.

6.3.1. Splitting the Trajectories ("subfiling")

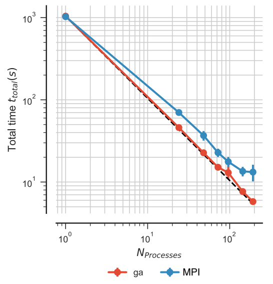
Subfiling is a mechanism previously used for splitting a large multi-dimensional global array to a number of smaller subarrays in which each smaller array is saved in a separate file. Subfiling reduces the file system control overhead by decreasing the number of processes concurrently accessing a shared file [72, 73]. Because subfiling is known to improve programming flexibility and performance of parallel shared-file I/O, we investigated splitting our trajectory file into as many trajectory segments as the number of processes. The trajectory file was split into N segments, one for each process, with each segment having N_b frames. This way, each process would access its own trajectory segment file without competing for file accesses.

We ran a benchmark up to 8 nodes (192 cores) and observed rather better scaling behavior with efficiencies above 0.6 (Figure 5b and 5c) with the delay time for stragglers reduced from 65 s to about 10 s for 72 processes. However, scaling was still far from ideal due to the MPI communication costs. Although the delay due to communication was much smaller than compared to parallel RMSD with shared-file I/O [compare Figure 5d ($t_{\text{comm}}^{\text{Straggler}} \gg t_{\text{comp}} + t_{\text{I/O}}$) to Figure 2d ($t_{\text{comm}}^{\text{Straggler}} \approx t_{\text{comp}} + t_{\text{I/O}}$)], it was still delaying several processes and resulted in longer job completion times (Figure 5d). These delayed tasks impacted performance so that speed-up remained far from ideal (Figure 5c).

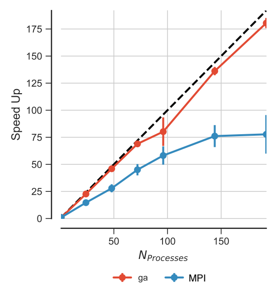
The subfiling approach appeared promising but it required preprocessing of trajectory files and additional storage space for the segments. We benchmarked the necessary time for splitting the trajectory in parallel using different number of MPI processes



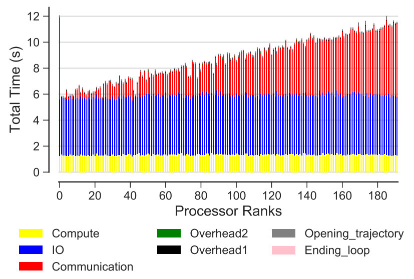
(a) Scaling for different components



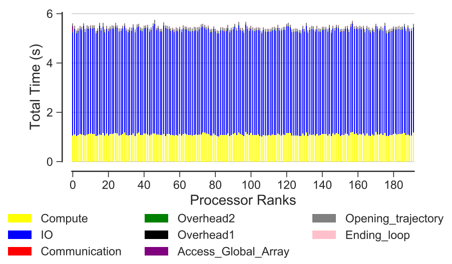
(b) Scaling total



(c) Speed-up



(d) Time comparison on different parts of the calculations per MPI rank with MPI collective communications.



(e) Time comparison on different parts of the calculations per MPI rank using *Global Arrays*

Figure 5: Comparison of the performance of the RMSD task on *SDSC Comet* when the trajectories are split (*subfiling*). The communication step used either MPI collective communications ("MPI", with `MPI_Gather()`) or *Global Arrays* ("ga", as described in Section 6.4). In the case of Global Arrays, all ranks updated the global RMSD array (`ga_put()`) and rank 0 accessed the whole RMSD array through the global memory address (`ga_get()`). Five repeats were performed to collect statistics. (a) Compute and I/O scaling versus number of processes. In serial, there is no communication and no data points are shown for $N = 1$. (b) Total time scaling versus number of processes. (c) Speed-up. (a-c) The error bars show standard deviation with respect to the mean. (d-e) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, access to the whole global RMSD array by rank 0, $t_{\text{Access\_Global\_Array}}$, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for the definitions.

(Table 5); in general the operation scaled well, with efficiencies > 0.8 although performance fluctuated, as seen for the case on six nodes where the efficiency dropped to 0.34 for the run. These preprocessing times were not included in the estimates because we focused on better understanding the principal causes of stragglers and we wanted to make the results directly comparable to the results of the previous sections. Nevertheless, from an end user perspective, preprocessing of trajectories can be integrated in workflows (especially as the data in Table 5 indicated good scaling) and the preprocessing time can be quickly amortized if the trajectories are analyzed repeatedly.

N_{seg}	N_{p}	nodes	time (s)	S	E
24	24	1	89.9	1.0	1.0
48	48	2	46.8	1.9	0.96
72	72	3	33.7	2.7	0.89
96	96	4	25.1	3.6	0.89
144	144	6	43.7	2.1	0.34
192	192	8	13.5	6.7	0.83

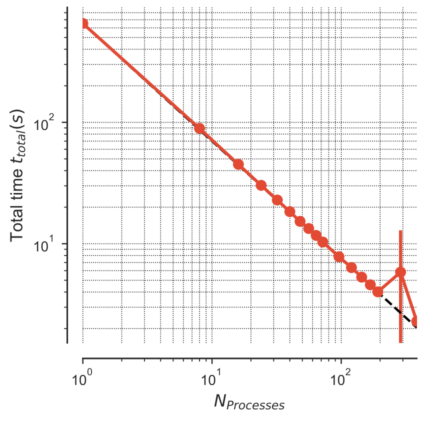
Table 5: The wall-clock time spent for writing N_{seg} trajectory segments using N_{p} processes using MPI on *SDSC Comet*. One set of runs was performed for the timings. Scaling S and efficiency E are relative to the 1 node case (24 MPI processes).

Often trajectories from MD simulations on HPC machines are produced and kept in smaller files or segments that can be concatenated to form a full continuous trajectory file. These trajectory segments could be used for the subfiling approach. However, it might not be feasible to have exactly one segment per MPI rank, with all segments of equal size, which constitutes the ideal case for subfiling. MDAnalysis can create virtual trajectories from separate trajectory segment files that appear to the user as a single trajectory. In Appendix C we investigated if this so-called *ChainReader* functionality could benefit from the subfiling approach. We found some improvements in performance but discovered limitations in the design of the ChainReader (namely that all segment files are initially opened) that will have to be addressed before equivalent performance can be reached.

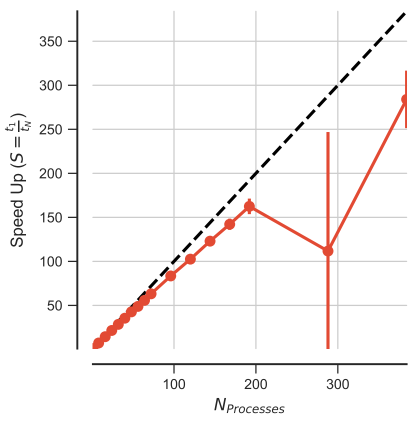
6.3.2. MPI-based Parallel HDF5

In the HPC community, parallel I/O with MPI-IO is widely used in order to address I/O limitations. We investigated MPI-based Parallel HDF5 to improve I/O scaling. We converted our XTC trajectory file into a simple custom HDF5 format so that we could test the performance of parallel I/O with the HDF5 file format. The code for this file format conversion can be found in the GitHub repository. The time it took to convert our XTC file with $2,512,200$ frames into HDF5 format was about 5,400 s on a local workstation with network file system (NFS).

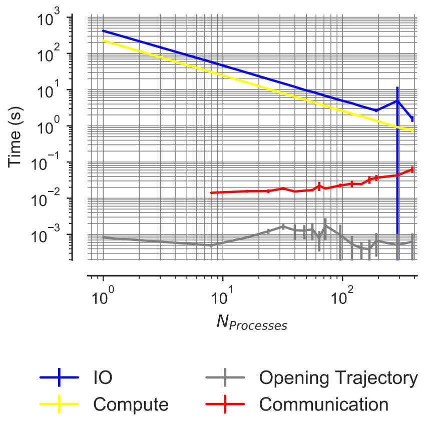
We ran our benchmark on up to 16 nodes (384 cores) on *SDSC Comet* and we observed near ideal scaling behavior Figures 6a and 6b) with parallel efficiencies above 0.8 on up to 8 nodes (Figure A.13a) with no straggler tasks (Figure 6d). The trajec-



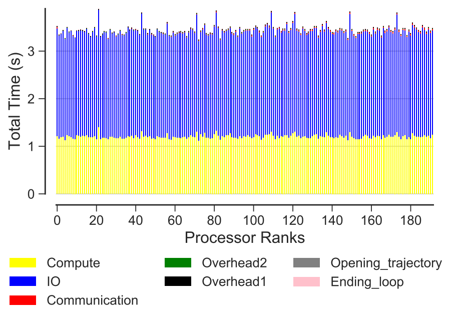
(a) Scaling total



(b) Speed-up



(c) Scaling for different components



(d) Time comparison on different parts of the calculations per MPI rank

Figure 6: Performance of the RMSD task with MPI-based parallel HDF5 (MPI-IO) on *SDSC Comet*. Data are read from the file system from a shared HDF5 file format instead of XTC format (independent I/O) and results are communicated back to rank 0. Five repeats were performed to collect statistics. (a-c) The error bars show standard deviation with respect to the mean. In serial, there is no communication and no data points are shown for $N = 1$ in (c). (d) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for definitions. These are typical data from one run of the five repeats.

tory reading I/O ($t_{\text{I/O}}$) was the dominant contribution, followed by compute (t_{comp}), but because both contributions scaled well, overall scaling performance remained good, even for 384 cores. We observed a low-performing outlier for 12 nodes (288 cores) with slower I/O than typical but did not further investigate. Importantly, the trajectory opening cost remained negligible (in the millisecond range) and the cost for MPI communication also remained small (below 0.1 s, even for 16 nodes). Overall, parallel MPI with HDF5 appeared to be a robust approach to obtain good scaling, even for I/O-bound tasks.

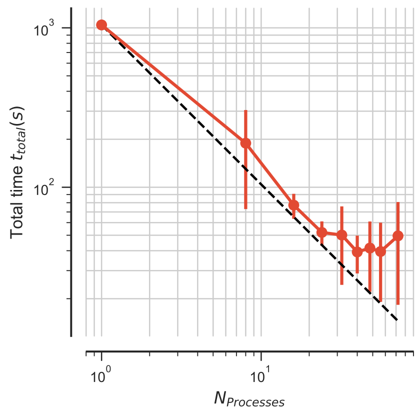
6.4. *Testing the Global Arrays Toolkit*

The *Global Arrays* (GA) toolkit [33] is a convenient layer to represent and access arrays across multiple MPI ranks and nodes. Because of its convenience and possibly reduced communications overhead due to its use of shared memory on a physical node and MPI for inter-node communication (see Section 3.3) we wanted to compare parallel trajectory analysis with GA to the MPI-based implementation that was discussed in the previous sections.

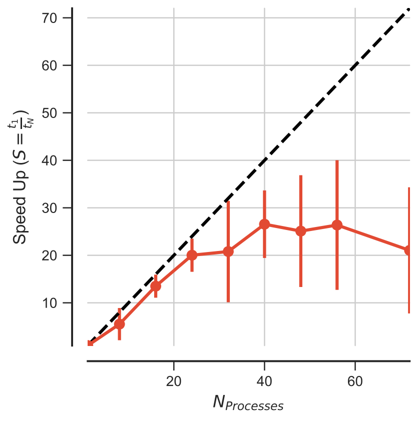
With GA, one large RMSD array called the *global array* was defined and each MPI rank updated its associated block in the global RMSD array using `ga_put()` (Algorithm 2). At the end, when all the processes exited the `block_rmsd()` function and updated their local block in the global array, rank 0 accessed the whole global array using `ga_access()`. In GA, the time for communication is $t_{\text{ga\_put()}} + t_{\text{ga\_access()}}$. We tested that the approach with GA (Algorithm 2) gave the same results as the previously discussed approach with `MPI_Gather()` (Algorithm 1).

Shared file. Using GA improved the strong scaling performance (Figures 7a and 7b) by reducing the communication time. Nevertheless, the remaining variation in the trajectory I/O part of the calculation and in particular the initial opening of the trajectory prevented ideal scaling (Figure 7c). Stragglers were primarily due to the fact that all ranks had to open the same trajectory file at the beginning of the execution (Figure 7d). In this case, these slow processes took about 50 s, which was slower than the mean execution time of all other ranks of 17 s. Trajectory opening was already problematic in the initial test (Figure 2c), which was still dominated by the communication cost. By substantially reducing communication cost, the simultaneous trajectory opening by multiple ranks emerged as the next dominant cause for stragglers.

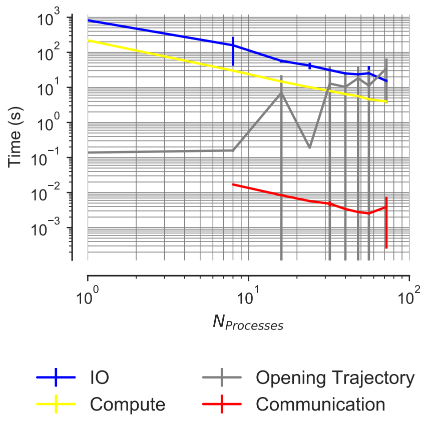
Subfiling. We tested subfiling (see Section 6.3.1) with GA to reduce the initial delay due to trajectory opening. Under otherwise identical conditions as in the previous sec-



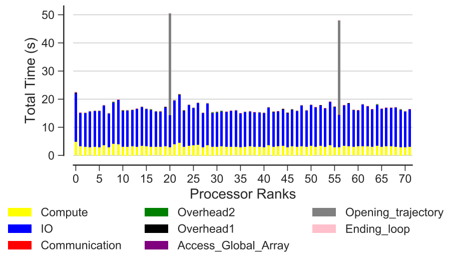
(a) Scaling total



(b) Speed-up



(c) Scaling for different components



(d) Time comparison on different parts of the calculations per MPI rank

Figure 7: Performance of the RMSD task using *Global Arrays* on *SDSC Comet*. All ranks updated the global RMSD array (`ga_put()`) and rank 0 accesseed the whole RMSD array through the global memory address (`ga_get()`). Five repeats were performed to collect statistics. (a-c) The error bars show standard deviation with respect to the mean. In serial, there is no communication and not data points are shown for $N = 1$ in (c). In (d), compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, access to the whole global array by rank 0, $t_{\text{Access\_Global\_Array}}$, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank are shown; see Table 3 for definitions. These are typical data from one run of the five repeats. MPI ranks 20 and 56 were stragglers.

tion we now observed near ideal scaling behavior with efficiencies above 0.9 (Figure 5b and 5c) without any straggler tasks (Figure 5e). Although the reason why in our case GA appeared to be more efficient than direct MPI-based communication remained unclear, these results showed that contention for file access clearly impacted performance.

By removing the contention, near ideal scaling could be achieved.

6.5. Likely Causes of Stragglers

The data indicated that an increase in the duration of both MPI communication and trajectory file access lead to large variability in the run time of individual MPI processes and ultimately poor scaling performance beyond a single node. A discussion of likely causes for stragglers begins with the observation that opening and reading a single trajectory file from multiple MPI processes appeared to be at the center of the problem.

In MDAnalysis, individual trajectory frames are loaded into memory, which ensures that even systems with tens of millions of atoms can be efficiently analyzed on resources with moderate RAM sizes. The test trajectory (file size 30 GB) had $2,512,200$ frames in total so each frame was about 0.011 MB in size. With $t_{\text{I/O}} \approx 0.3$ ms per frame, the data were ingested at a rate of about 40 MB/s for a single process. For 24 MPI ranks (one *SDSC Comet* node), the aggregated reading rate would have been about 1 GB/s, well within the available bandwidth of 56 Gb/s of the InfiniBand network interface that served the Lustre file system, but nevertheless sufficient to produce substantial constant network traffic.

Furthermore, in our study the default Lustre stripe size value was 1 MB, i.e., the amount of contiguous data stored on a single Lustre object storage target (OST). Each I/O request read a single Lustre stripe but because the I/O size (0.011 MB) was smaller than the stripe size, many of these I/O requests were likely just accessing the same stripe on the same OST but nevertheless had to acquire a new reading lock for each request. The reason for this behavior is related to ensuring POSIX consistency that relates to POSIX I/O API and POSIX I/O semantics, which can have adverse effects on scalability and performance. Parallel file systems like Lustre implement sophisticated distributed locking mechanisms to ensure consistency. For example, locking mechanisms ensures that a node can not read from a file or part of a file while it might be being modified by another node. In fact, when the application I/O is not designed in a way to avoid scenarios where multiple nodes are fighting over locks for overlapping extents, Lustre can suffer from scalability limitations [74]. Continuously keeping metadata updated in order to have fully consistent reads and writes (POSIX metadata management), requires writing a new value for the file's last-accessed time (POSIX atime) every time a file is read, imposing a significant burden on parallel file system [75]. It was observed that contention for the interconnect between OSTs and compute nodes due to MPI communication may lead to variable performance in I/O measurements [76]. Conversely, our data suggest that single-shared-file I/O on Lustre can negatively affect MPI communication as well, even at moderate numbers (tens to hundreds) of concurrent requests, similar to recent network simulations that predicted interference between MPI and I/O traffic [77]. This work indicated that MPI traffic (inter-process communication) can be affected by increasing I/O, and in particular, a few MPI processes were always delayed by one to two orders of magnitude more than the median time. In summary, these observations in the context of the work by Brown et al. [77] suggest that our observed stragglers with large variance in the communication step might be due to interference of MPI communications with the I/O requests on the same network.

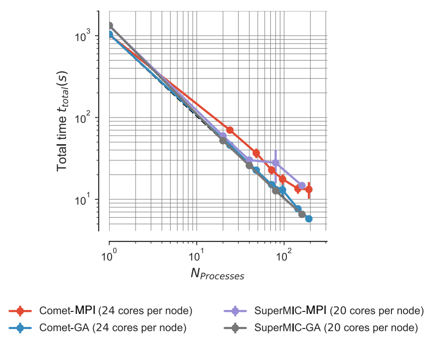
7. Reproducibility and Performance Comparison on Different Clusters

In this section we compare the performance of the RMSD task on different HPC resources (Table 1) to examine the robustness of the methods we used for our performance study and to ensure that the results are general and independent from the specific HPC system. Scripts and instructions to set up the computational environments and reproduce our computational experiments are provided in a git repository as described in section 5.

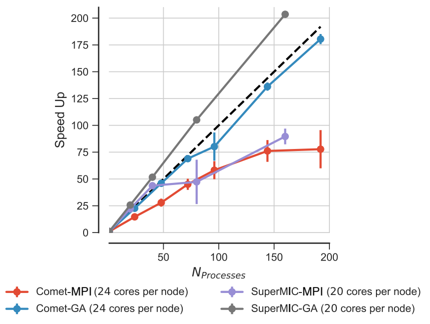
In Appendix A, we demonstrated that stragglers occur on *PSC Bridges* (Figure A.11) and *LSU SuperMIC* (Figure A.12) in a manner similar to the one observed on *SDSC Comet* (section 6.1). We performed additional comparisons for several cases discussed previously, namely (1) splitting the trajectories with blocking collective communications in MPI, (2) splitting the trajectories with Global Arrays for communications, and (3) MPI-based parallel HDF5.

7.1. Splitting the Trajectories

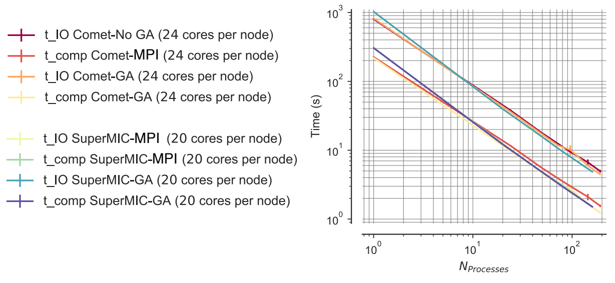
Figure 8 shows the strong scaling of the RMSD task on different HPC resources. Splitting the trajectories with Global Arrays for communication resulted in very good scaling performance on *LSU SuperMIC*, similar to the results obtained on *SDSC Comet*. The results with MPI blocking collective communication (instead of Global Arrays) were also comparable between the two clusters, with scaling far from ideal due to the communication cost (see section 6.3.1 and Figures 5d and A.14). Overal, the scaling of the RMSD task is better on *LSU SuperMIC* than on *SDSC Comet* and the performance gap increased with increasing core number. The results on *LSU SuperMIC* confirmed the conclusion obtained on *SDSC Comet* that at least in this case Global Arrays performed better than MPI blocking collective communication.



(a) Scaling total



(b) Speed-up



(c) Scaling of t_{comp} and $t_{\text{I/O}}$.

Figure 8: Comparison of the performance of the RMSD task across different clusters (*SDSC Comet*, *LSU SuperMIC*) when the trajectories are split (*subfiling*). Results were communicated back to rank 0 either with MPI collective communications (label "MPI") or using *Global Arrays* (label "GA"). Five repeats were performed to collect statistics. The error bars show the standard deviation with respect to the mean.

7.2. MPI-based Parallel HDF5

Figure 9 shows the scaling on *SDSC Comet*, *LSU SuperMIC*, and *PSC Bridges* using MPI-based parallel HDF5. Performance on *SDSC Comet* and *LSU SuperMIC* was very good with near ideal linear strong scaling. The performance on *PSC Bridges* was sensitive to how many cores per node were used. Using all 28 cores in a node resulted in poor performance but decreasing the number of cores per node and equally distributing processes over nodes improved the scaling (Figure 9), mainly by reducing variation in the I/O times.

The main difference between the runs on *PSC Bridges* and *SDSC Comet/LSU SuperMIC* appeared to be the variance in $t_{\text{I/O}}$ (Figure 9c). The I/O time distribution was fairly small and uniform across all ranks on *SDSC Comet* and *LSU SuperMIC* (Figures 10b and 6d). However, on *PSC Bridges* the I/O time was on average about two and a half times larger and the I/O time distribution was also more variable across different ranks (Figure 10a).

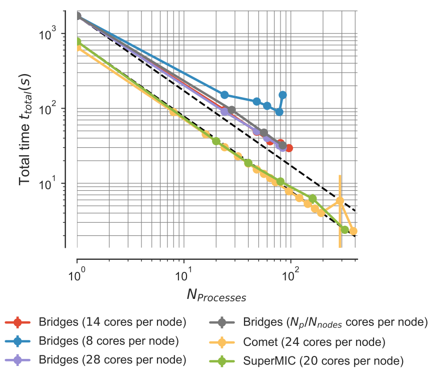
7.3. Comparison of Compute and I/O Scaling Across Different Clusters

A full comparison of compute and I/O scaling across different clusters for different test cases and algorithms is shown in Table 6. For MPI-based parallel HDF5, both the compute and I/O time on *Bridges* were consistenly larger than their corresponding values on *SDSC Comet* and *LSU SuperMIC*. For example, with one core the corresponding compute and I/O time were t_{comp} = 387 s, $t_{\text{I/O}}$ = 1318 s versus 225 s, 423 s on *SDSC Comet* and 273 s, 503 s on *LSU SuperMIC*. This performance difference became larger with increasing core number. When the trajectories were split and Global Arrays was used for communication both *SDSC Comet* and *LSU SuperMIC* showed similar performance.

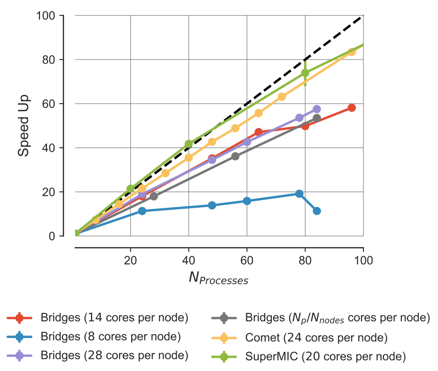
Overall, the results from *SDSC Comet* and *LSU SuperMIC* are consistent with each other. Performance on *PSC Bridges* seemed sensitive to the exact allocation of cores on each node but nevertheless the approaches that decreased the occurence of stragglers on *SDSC Comet* and *LSU SuperMIC* also improved performace on *PSC Bridges*. Thus, the findings described in the previous sections are valid for a range of different HPC clusters with Lustre file systems.

				$N_{Processes}$							
Cluster	**Gather**	**File Access**	**Time**	**Serial**	**Comet: 24 Bridges: 24 SuperMIC: 20**	**Comet: 48 Bridges: 48 SuperMIC: 40**	**Comet: 72 Bridges: 60 SuperMIC: 80**	**Comet: 96 Bridges: 78**	**Comet: 144 Bridges: 84 SuperMIC: 160**	**Comet: 192**	**Comet: 384 SuperMIC: 320**
Comet	MPI	Single	$t_{\text{I/O}}$	791 ± 5.22	49 ± 3.45	29 ± 1.3	26 ± 9.19	–	–	–	–
			t_{comp}	225 ± 5.4	11 ± 0.75	6 ± 0.35	4 ± 0.48				
Bridges	MPI	Single	$t_{\text{I/O}}$	770 ± 10.8	38 ± 0.84	33 ± 19.4	15 ± 1.6	–	–	–	–
			t_{comp}	221 ± 3.9	11 ± 0.43	6 ± 0.32	4 ± 0.18				
SuperMIC	MPI	Single	$t_{\text{I/O}}$	1014.51 ± 2.94	48.08 ± 0.35	24.5 ± 0.79	12 ± 0.31	–	6.24 ± 0.38	–	–
			t_{comp}	303.85 ± 2.3	14.56 ± 0.14	7.4 ± 0.25	3.7 ± 0.12		1.8 ± 0.04		
Comet	GA	Single	$t_{\text{I/O}}$	820 ± 18.49	41 ± 8.99	23 ± 4.14	15 ± 2.06	–	–	–	–
			t_{comp}	219 ± 9.8	10 ± 0.3	5 ± 0.48	3 ± 0.54				
Comet	MPI	Splitting	$t_{\text{I/O}}$	799 ± 5.22	37 ± 1.22	18 ± 0.18	12 ± 0.14	9 ± 0.3	6 ± 0.66	4 ± 0.23	–
			t_{comp}	225 ± 5.4	11 ± 0.31	5 ± 0.07	3 ± 0.04	3 ± 0.11	2 ± 0.23	1 ± 0.07	
SuperMIC	MPI	Splitting	$t_{\text{I/O}}$	1013.75 ± 2.8	39.99 ± 0.36	19.18 ± 0.25	9.61 ± 0.28	–	4.83 ± 0.06	–	–
			t_{comp}	304.26 ± 2.55	12.41 ± 0.22	5.99 ± 0.09	3.08 ± 0.13		1.5 ± 0.01		
Comet	GA	Splitting	$t_{\text{I/O}}$	820 ± 18.5	36 ± 0.78	17 ± 0.3	11 ± 0.23	10 ± 1.7	5 ± 0.14	4 ± 0.07	–
			t_{comp}	219 ± 9.5	9 ± 0.22	4 ± 0.07	3 ± 0.04	2 ± 0.4	1 ± 0.05	1 ± 0.02	
SuperMIC	GA	Splitting	$t_{\text{I/O}}$	1027.62 ± 10.32	39.62 ± 0.2	19.66 ± 0.1	9.57 ± 0.1	–	4.86 ± 0.05	–	–
			t_{comp}	305.78 ± 3.47	12.16 ± 0.1	6.01 ± 0.007	2.97 ± 0.1		1.51 ± 0.03		
Comet	MPI	PHDF5	$t_{\text{I/O}}$	423 ± 5.88	19 ± 0.3	9 ± 0.13	6 ± 0.06	5 ± 0.12	3 ± 0.2	3 ± 0.25	1.57 ± 0.29
			t_{comp}	225 ± 6.55	10 ± 0.12	5 ± 0.1	3 ± 0.04	2 ± 0.05	1 ± 0.04	1 ± 0.03	0.76 ± 0.09
Bridges	MPI	PHDF5	$t_{\text{I/O}}$	1318.87 ± 10.42	67.93 ± 0.52	37.37 ± 0.2	30.35 ± 0.15	24.16 ± 0.89	22.5 ± 0.17	–	–
			t_{comp}	387.8 ± 5.51	21.97 ± 0.38	12.12 ± 0.34	9.79 ± 0.24	7.72 ± 0.03	7.18 ± 0.08		
SuperMIC	MPI	PHDF5	$t_{\text{I/O}}$	503.69 ± 2.57	12.96 ± 0.06	6.46 ± 0.02	3.2 ± 0.01	–	1.64 ± 0.01	–	0.82 ± 0.004
			t_{comp}	273.54 ± 4.7	23.44 ± 0.29	12.22 ± 0.43	7.3 ± 0.85		4.59 ± 0.96		1.55 ± 0.009

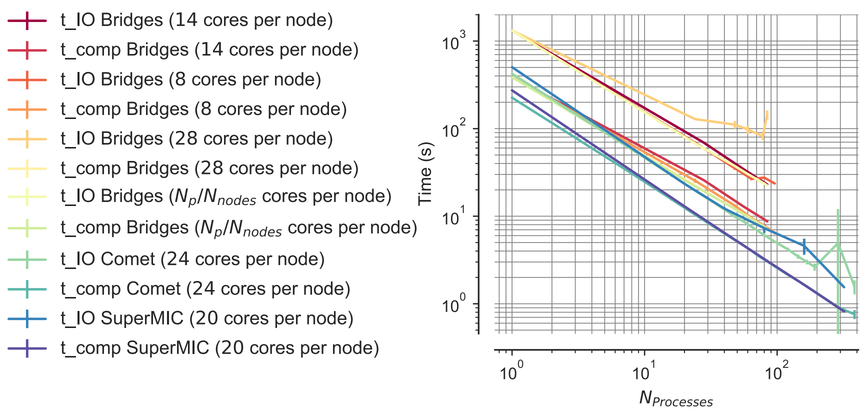
Table 6: Comparison of the compute and I/O scaling for different test cases and number of processes. Five repeats were performed to collect statistics. The mean value and the standard deviation with respect to mean are reported for each case.



(a) Scaling total



(b) Speed-up

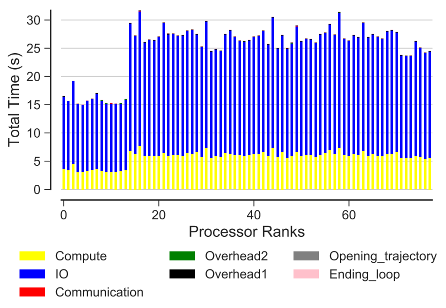


(c) Scaling of t_{comp} and $t_{\text{I/O}}$

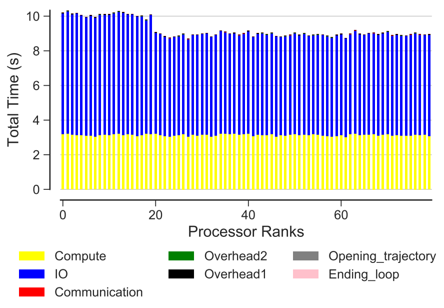
Figure 9: Comparison of the performance of the RMSD task across different clusters (*SDSC Comet*, *PSC Bridges*, *LSU SuperMIC*) with MPI-IO. Data were read from a shared HDF5 file instead of an XTC file, using MPI independent I/O in the PHDF5 library. Results were communicated back to rank 0. $N_{\text{P}}/N_{\text{nodes}}$ indicates that number of processes used for the task were equally distributed over all compute nodes. Five repeats were performed to collect statistics. The error bars show standard deviation with respect to mean. In (b) only results up to 100 cores are shown to simplify the comparison; see Fig. 6b for *SDSC Comet* and Fig. A.13c for *LSU SuperMic* data.

8. Guidelines for Improving Parallel Trajectory Analysis Performance

Although the performance measurements were performed with *MDAnalysis* and therefore capture some details of this library such as the specific timings for file reading, we believe that the broad picture is fairly general and applies to any Python-based approach that uses MPI for parallelizing trajectory access with a split-apply-combine approach. Based on the lessons that we learned, we suggest the following guidelines



(a) *PSC Bridges*



(b) *LSU SuperMIC*

Figure 10: Examples of timing per MPI rank for the RMSD task with MPI-based parallel HDF5 on (a) *PSC Bridges* and (b) *LSU SuperMIC*. Five repeats were performed to collect statistics and these were typical data from one run of the five repeats. Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for definitions.

to improve strong scaling performance:

Heuristic 1 Calculate compute to I/O ratio ($R_{\text{comp/IO}}$, Eq. 7) and compute to communication ratio ($R_{\text{comp/comm}}$, Eq. 8). $R_{\text{comp/IO}}$ determines whether the task is compute bound ($R_{\text{comp/IO}} \gg 1$) or IO bound ($R_{\text{comp/IO}} \ll 1$). $R_{\text{comp/comm}}$ determines whether the task is communication bound ($\frac{\overline{t_{\text{comp}}}}{\overline{t_{\text{comm}}}} \ll 1$) or compute bound ($R_{\text{comp/IO}} \gg 1$).

As discussed in Section 6.2, for I/O bound problems the interference between MPI communication and I/O traffic can be problematic [50, 77] and the performance of the task will be affected by the straggler tasks that delay job completion time.

Heuristic 2 For $R_{\text{comp/IO}} \geq 1$, single-shared-file I/O can be used and will not decrease performance. One may consider the following cases:

Heuristic 2.1 If $R_{\text{comp/comm}} \gg 1$, the task is compute bound and will scale well as shown in Figure 3.

Heuristic 2.2 If $R_{\text{comp/comm}} \ll 1$, one might consider using *Global Arrays* to improve scaling by utilizing efficient distribution of data via the shared arrays (section 6.4).

Heuristic 3 For $R_{\text{comp/IO}} \leq 1$ the task is I/O bound and single-shared-file I/O should be avoided to remove unnecessary metadata operations. One might want to consider the following steps:

Heuristic 3.1 If there is access to HDF5 format, use MPI-based Parallel HDF5 (Section 6.3.2).

Heuristic 3.2 If the trajectory file is not in HDF5 format then one can consider subfiling and split the single trajectory file into as many trajectory segments as the number of processes. Splitting the trajectories can be easily performed in parallel and trajectory conversion may be integrated into the beginning of standard workflows for MD simulations. Alternatively, trajectories may be kept in smaller chunks if they are already produced in batches; for instance, when running simulations with *Gromacs* [55], the `gmx mdrun -noappend` option produces individual trajectory segments instead of extending an existing trajectory file.

Heuristic 3.3 In case of $R_{\text{comp/comm}} \ll 1$, use of *Global Arrays* may be considered to potentially improve scaling (Section 6.3.1).

9. Conclusions

We analyzed the strong scaling performance of a typical task when analysing MD trajectories, the calculation of the time series of the RMSD of a protein, with the widely used Python-based *MDAnalysis* library. The task was parallelized with MPI following the *split-apply-combine* approach by having each MPI process analyze a contiguous segment of the trajectory. This approach did not scale beyond a single node because straggler MPI processes exhibited large upward variations in runtime. Stragglers were primarily caused by either increased MPI communication costs or increased time to open the single shared trajectory file whereas both the computation and the ingestion of data exhibited close to ideal strong scaling behavior. Stragglers were less prevalent for compute-bound workloads (i.e., $R_{\text{comp/IO}} \gg 1$), suggesting that file read I/O was responsible for poor MPI communication. In particular, artifically removing all I/O substantially improved performance of the communication step and thus brought overall performance close to ideal (i.e., linear increase in speed-up with processor count with slope one). By performing benchmarks on three different XSEDE supercomputers we showed that our results were independent from the specifics of the hardware and local environment. Our results hint at the possibility that stragglers might be due to the competition between MPI messages and the Lustre file system on the shared InfiniBand interconnect, which would be consistent with other similar observations [50] and theoretical predictions by Brown et al. [77]. One possible interpretation of our results is that for a sufficiently large per-frame compute workload, read I/O interferes much less with communication than for an I/O bound task that almost continuously accesses the file system. This intepretation suggested that we needed to improve read I/O to reduce interference.

We investigated subfiling (splitting of the trajectories into separate files, one for each MPI rank) and MPI-based parallel I/O. Subfiling improved scaling, especially when combined with the *Global Arrays* toolkit. *Global Arrays* reduced the communication cost compared to MPI collective communications even though it only acts as programming layer to access data across multiple nodes in a convenient array form and also uses MPI for its inter-node data exchange. Subfiling with *Global Arrays* achieved nearly ideal scaling up to 192 cores (8 nodes on *SDSC Comet*). When we used MPI-based parallel I/O through HDF5 together with MPI for communications we achieved nearly ideal performance up to 384 cores (16 nodes on *SDSC Comet*) and speed-ups of two orders of magnitude compared to the serial execution. The latter approach appears to be a promising way forward as it directly builds on very widely used technology (MPI-IO and HDF5) and echoes the experience of the wider HPC community that parallel file I/O is necessary for efficient data handling.

The biomolecular simulation community suffers from a large number of trajectory file formats with very few being based on HDF5, with the exception of the H5MD format [78] and the MDTraj HDF5 format [20]. Our work suggests that HDF5-based formats should be seriously considered as the default for MD simulations if users want to make efficient use of their HPC systems for analysis. Alternatively, enabling MPI-IO for trajectory readers in libraries such as *MDAnalysis* might also provide a path forward to better read performance.

We summarized our findings in a number of guidelines for improving the scaling of parallel analysis of MD trajectory data. We showed that it is feasible to run an I/O bound analysis task on HPC resources with a Lustre parallel file system and achieve good scaling behavior up to 384 CPU cores with an almost 300-fold speed-up compared to serial execution. Although we focused on the *MDAnalysis* library, similar strategies are likely to be more generally applicable and useful to the wider biomolecular simulation community.

Acknowledgements

We are grateful to Sarp Oral for insightful comments on this manuscript. This work was supported by the National Science Foundation under grant numbers ACI-1443054 and ACI-1440677. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. *SDSC Comet* at the San Diego Supercomputer Center, *LSU SuperMic* at Louisiana State University, and *PSC Bridges* at the Pittsburgh Supercomputing Center were used under allocations TG-MCB090174 and TG-MCB130177.

References

1. D. W. Borhani, D. E. Shaw, The future of molecular dynamics simulations in drug discovery, J Comput Aided Mol Des 26 (2012) 15–26. doi:10.1007/s10822-011-9517-y.
2. R. O. Dror, R. M. Dirks, J. P. Grossman, H. Xu, D. E. Shaw, Biomolecular simulation: a computational microscope for molecular biology, Annu Rev Biophys 41 (2012) 429–52. doi:10.1146/annurev-biophys-042910-155245.
3. M. Orozco, A theoretical view of protein dynamics, Chem. Soc. Rev. 43 (2014) 5051–5066. doi:10.1039/C3CS60474H.
4. J. R. Perilla, B. C. Goh, C. K. Cassidy, B. Liu, R. C. Bernardi, T. Rudack, H. Yu, Z. Wu, K. Schulten, Molecular dynamics simulations of large macromolecular complexes, Current Opinion in Structural Biology 31 (2015) 64 – 74. URL: http://www.sciencedirect.com/science/article/pii/S0959440X15000342. doi:http://dx.doi.org/10.1016/j.sbi.2015.03.007.

5. S. Bottaro, K. Lindorff-Larsen, Biophysical experiments and biomolecular simulations: A perfect match?, Science 361 (2018) 355–360. doi:10.1126/science.aat4010.
6. M. E. Tuckerman, Statistical Mechanics: Theory and Molecular Simulation, Oxford University Press, Oxford, UK, 2010.
7. C. Mura, C. E. McAnany, An introduction to biomolecular simulations and docking, Molecular Simulation 40 (2014) 732–764. URL: http://dx.doi.org/10.1080/08927022.2014.935372. doi:10.1080/08927022.2014.935372.
8. T. Cheatham, D. Roe, The impact of heterogeneous computing on workflows for biomolecular simulation and analysis, Computing in Science Engineering 17 (2015) 30–39. doi:10.1109/MCSE.2015.7.
9. G. R. Kneller, V. Keiner, M. Kneller, M. Schiller, nmoldyn: A program package for a neutron scattering oriented analysis of molecular dynamics simulations, Computer Physics Communications 91 (1995) 191 – 214. URL: http://www.sciencedirect.com/science/article/pii/001046559500048K. doi:http://dx.doi.org/10.1016/0010-4655(95)00048-K.
10. K. Hinsen, E. Pellegrini, S. Stachura, G. R. Kneller, nmoldyn 3: Using task farming for a parallel spectroscopy-oriented analysis of molecular dynamics simulations, Journal of Computational Chemistry 33 (2012) 2043–2048. URL: http://dx.doi.org/10.1002/jcc.23035. doi:10.1002/jcc.23035.
11. W. Humphrey, A. Dalke, K. Schulten, VMD – Visual Molecular Dynamics, J. Mol. Graph. 14 (1996) 33–38. URL: http://www.ks.uiuc.edu/Research/vmd/.
12. K. Hinsen, The molecular modeling toolkit: a new approach to molecular simulations, Journal of Computational Chemistry 21 (2000) 79–85.
13. B. J. Grant, A. P. C. Rodrigues, K. M. ElSawy, J. A. McCammon, L. S. D. Caves, Bio3d: an r package for the comparative analysis of protein structures, Bioinformatics 22 (2006) 2695–6. doi:10.1093/bioinformatics/btl461.

14. T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, D. E. Shaw, A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories, in: 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, pp. 1–12. doi:10.1109/SC.2008.5214715.

15. T. D. Romo, A. Grossfield, LOOS: An extensible platform for the structural analysis of simulations, in: 31st Annual International Conference of the IEEE EMBS, IEEE, Minneapolis, Minnesota, USA, 2009, pp. 2332–2335.

16. T. D. Romo, N. Leioatts, A. Grossfield, Lightweight object oriented structure analysis: Tools for building tools to analyze molecular dynamics simulations, Journal of Computational Chemistry 35 (2014) 2305–2318. URL: http://dx.doi.org/10.1002/jcc.23753. doi:10.1002/jcc.23753.

17. N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, O. Beckstein, MDAnalysis: A toolkit for the analysis of molecular dynamics simulations, J Comp Chem 32 (2011) 2319–2327. doi:10.1002/jcc.21787.

18. R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domański, S. Buchoux, I. M. Kenney, O. Beckstein, MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations, in: S. Benthall, S. Rostrup (Eds.), Proceedings of the 15th Python in Science Conference, SciPy, Austin, TX, 2016, pp. 102 – 109. URL: http://mdanalysis.org.

19. D. R. Roe, I. Thomas E. Cheatham, Ptraj and cpptraj: Software for processing and analysis of molecular dynamics trajectory data, Journal of Chemical Theory and Computation 9 (2013) 3084–3095. URL: http://dx.doi.org/10.1021/ct400341p. doi:10.1021/ct400341p, pMID: 26583988.

20. R. T. McGibbon, K. A. Beauchamp, M. P. Harrigan, C. Klein, J. M. Swails, C. X. Hernández, C. R. Schwantes, L.-P. Wang, T. J. Lane, V. S. Pande, MDTraj: A modern open library for the analysis of molecular dynamics trajectories, Biophysical Journal 109 (2015) 1528 – 1532. URL: http://www.sciencedirect.com/science/article/pii/

S0006349515008267. doi:10.1016/j.bpj.2015.08.015.

21. S. O. Yesylevskyy, Pteros 2.0: Evolution of the fast parallel molecular analysis library for c++ and python, Journal of Computational Chemistry 36 (2015) 1480–1488. URL: http://dx.doi.org/10.1002/jcc.23943. doi:10.1002/jcc.23943.
22. S. Doerr, M. J. Harvey, F. Noé, G. De Fabritiis, HTMD: High-throughput molecular dynamics for molecular discovery, Journal of Chemical Theory and Computation 12 (2016) 1845–1852. URL: http://dx.doi.org/10.1021/acs.jctc.6b00049. doi:10.1021/acs.jctc.6b00049.
23. M. Khoshlessan, I. Paraskevakos, S. Jha, O. Beckstein, Parallel analysis in MDAnalysis using the Dask parallel computing library, in: Katy Huff, David Lippa, Dillon Niederhut, M. Pacer (Eds.), Proceedings of the 16th Python in Science Conference, SciPy, Austin, TX, 2017, pp. 64–72. doi:10.25080/shinma-7f4c6e7-00a.
24. I. Paraskevakos, A. Luckow, M. Khoshlessan, G. Chantzialexiou, T. E. Cheatham, O. Beckstein, G. Fox, S. Jha, Task-parallel analysis of molecular dynamics trajectories, in: ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA, Association for Computing Machinery, ACM, New York, NY, USA, 2018, p. Article No. 49.
25. P. Liu, D. K. Agrafiotis, D. L. Theobald, Fast determination of the optimal rotational matrix for macromolecular superpositions, J Comput Chem 31 (2010) 1561–3. doi:10.1002/jcc.21439.
26. A. R. Leach, Molecular Modelling. Principles and Applications, Longman, 1996.
27. M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, in: Proceedings of the 14th Python in Science Conference, 2015, pp. 130–136. URL: https://github.com/dask/dask.
28. L. D. Dalcín, R. R. Paz, P. A. Kler, A. Cosimo, Parallel distributed computing using python, Advances in Water Resources 34 (2011) 1124 – 1139. doi:10.1016/j.advwatres.2011.04.013, new Computational Methods and Software Tools.

29. L. Dalcín, R. Paz, M. Storti, MPI for python, Journal of Parallel and Distributed Computing 65 (2005) 1108 – 1115. doi:10.1016/j.jpdc.2005.03.010.

30. P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters, IEEE Transactions on Services Computing 12 (2016) 91–104. doi:10.1109/TSC.2016.2611578.

31. J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, N. Wilkins-Diehr, XSEDE: Accelerating scientific discovery, Computing in Science & Engineering 16 (2014) 62–74. URL: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80. doi:10.1109/MCSE.2014.80.

32. J. A. DAILY, GAIN: DISTRIBUTED ARRAY COMPUTATION WITH PYTHON, Master's thesis, School of Electrical Engineering and Computer Science, WASHINGTON STATE UNIVERSITY, 2009.

33. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, applications and performance of the global arrays shared memory programming toolkit, The International Journal of High Performance Computing Applications 20 (2006) 203–231.

34. J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI'04 Sixth Symposium on Operating System Design and Implementation, 2004, pp. pp. 137–150.

35. G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 265–278.

36. J. Kyong, J. Jeon, S.-S. Lim, Improving scalability of apache spark-based scale-up server through docker container-based partitioning, in: Proceedings of the 6th International Conference on Software and Computer Applications - ICSCA '17, ACM Press, New York, USA, 2017, pp. 176–180. URL: http://dl.acm.org/citation.cfm?doid=3056662.3056686. doi:10.

1145/3056662.3056686.

37. K. Ousterhout, Architecting for Performance Clarity in Data Analytics Frameworks, Ph.D. thesis, EECS Department, University of California, Berkeley, Berkeley, CA, 2017. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-158.html. arXiv:https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-158.pdf.
38. A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy, M. W. Mahoney, Prabhat, Matrix factorizations at scale: A comparison of scientific data analytics in spark and c+mpi using three case studies, in: IEEE International Conference on Big Data (Big Data), 2016, pp. 204–213. URL: http://ieeexplore.ieee.org/document/7840606/. doi:10.1109/BigData.2016.7840606.
39. H. Yang, X. Liu, S. Chen, Z. Lei, H. Du, C. Zhu, Improving Spark performance with MPTE in heterogeneous environments, in: 2016 International Conference on Audio, Language and Image Processing (ICALIP), IEEE, 2016, pp. 28–33. URL: http://ieeexplore.ieee.org/document/7846627/. doi:10.1109/ICALIP.2016.7846627.
40. E. Schmidt, G. DeMichillie, F. Perry, T. Akidau, D. Halperin, Large-scale data analysis at cloud scale, in: Symposium on Frontiers in Big Data, 2016.
41. T.-D. Phan, Energy-efficient Straggler Mitigation for Big Data Applications on the Clouds, Ph.D. thesis, École normale supérieure de Renne, 2017.
42. Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, IEEE Transactions on Computers 63 (2014) 954–967. doi:10.1109/TC.2013.15.
43. B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, N. Podhorszki, Characterizing output bottlenecks in a supercomputer, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 8:1–8:11. URL: http://dl.acm.org/citation.cfm?id=2388996.2389007.

44. J. Rosen, B. Zhao, Fine-Grained Micro-Tasks for MapReduce Skew-Handling, Technical Report, EECS, UC Berkeley, 2012. URL: `https://pdfs.semanticscholar.org/3617/916adb83f33f8df7d0b3bfc23d0de80da9b7.pdf`.

45. Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune: Mitigating skew in mapreduce applications, pages 25-36, in: SIGMOD'12, SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. Pages 25–36. doi:`10.1145/2213836.2213840`.

46. K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks, in: NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, ISBN: 978-1-931971-218, 2015, pp. Pages 293–307.

47. B. Abdul-Wahid, H. Feng, D. Rajan, R. Costaouec, E. Darve, D. Thain, J. A. Izaguirre, Awe-wq, fast-forwarding molecular dynamics using the accelerated weighted ensemble, Journal of Chemical Information and Modeling 54 (2014) 3033–3043.

48. G. Wu, H. Song, D. Lin, A scalable parallel framework for microstructure analysis of large-scale molecular dynamics simulations data, Computational Materials Science 144 (2018) 322–330.

49. T. Tu, C. A. Rendleman, P. J. Miller, F. Sacerdoti, R. O. Dror, D. E. Shaw, Accelerating parallel analysis of scientific simulation data via zazen, in: 8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, http://www.usenix.org/events/fast10/tech/full_papers/tu.pdf, 2010, pp. 129–142.

50. J. E. Stone, B. Isralewitz, K. Schulten, Early experiences scaling vmd molecular visualization and analysis jobs on blue waters, in: Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013), IEEE Computer Society, Washington, DC, USA, 2013, pp. 43–50.

51. A. Shkurtia, R. Goni, P. Andrio, E. Breitmoserd, I. Bethuned, M. Orozco, C. A. Laughtona, pyPcazip: A PCA-based toolkit for compression and analysis of molecular simulation data, SoftwareX 5 (2016) 44–50.

52. P. Malakar, C. Knight, T. Munson, V. Vishwanath, M. E. Papka, Scalable in situ

analysis of molecular dynamics simulations, in: ISAV'17 Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization, 2017, pp. 1–6.

53. T. Johnston, B. Zhang, A. Liwo, S. Crivelli, M. Taufer, *In situ* data analytics and indexing of protein trajectories, J Comput Chem 38 (2017) 1419–1430. doi:10.1002/jcc.24729.

54. B. R. Brooks, C. L. Brooks III., A. D. J. Mackerell, L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, A. Caflisch, L. Caves, Q. Cui, A. R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kuczera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R. W. Pastor, C. B. Post, J. Z. Pu, M. Schaefer, B. Tidor, R. M. Venable, H. L. Woodcock, X. Wu, W. Yang, D. M. York, M. Karplus, CHARMM: the biomolecular simulation program., J. Comp. Chem. 30 (2009) 1545–1614. doi:10.1002/jcc.21287.

55. M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, E. Lindahl, GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers, SoftwareX 1–2 (2015) 19 – 25. doi:10.1016/j.softx.2015.06.001.

56. D. A. Case, T. E. Cheatham, 3rd, T. Darden, H. Gohlke, R. Luo, K. M. Merz, Jr, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, The amber biomolecular simulation programs, J Comput Chem 26 (2005) 1668–1688. doi:10.1002/jcc.20290.

57. J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, J Comput Chem 26 (2005) 1781–1802. doi:10.1002/jcc.20289.

58. S. K. Burley, H. M. Berman, C. Bhikadiya, C. Bi, L. Chen, L. D. Costanzo, C. Christie, J. M. Duarte, S. Dutta, et al., Protein Data Bank: the single global archive for 3D macromolecular structure data, Nucleic Acids Research 47 (2018) D520–D528. URL: http://dx.doi.org/10.1093/nar/gky949. doi:10.1093/nar/gky949.

59. S. Van Der Walt, S. C. Colbert, G. Varoquaux, The numpy array: a structure for efficient numerical computation, Computing in Science & Engineering 13

(2011) 22–30. doi:10.1109/MCSE.2011.37.

60. D. L. Theobald, Rapid calculation of RMSDs using a quaternion-based characteristic polynomial, Acta Crystallogr A 61 (2005) 478–80. doi:10.1107/S0108767305015266.

61. J. Daily, A. Vishnu, B. Palmer, H. van Dam, D. Kerbyson, On the suitability of MPI as a PGAS runtime, in: 2014 21st International Conference on High Performance Computing (HiPC), 2014, pp. 1–10. doi:10.1109/HiPC.2014.7116712.

62. J. Nieplocha, R. J. Harrison, R. J. Littlefield, Global arrays: A non-uniform-memory-access programming model for high-performance computers, Journal of Supercomputing 10 (1996) 169–189.

63. A. Collette, Python and hdf5, in: M. Blanchette, R. Roumeliotis (Eds.), Python and HDF5, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2014.

64. S. L. Seyler, O. Beckstein, Sampling of large conformational transitions: Adenylate kinase as a testing ground, Molec. Simul. 40 (2014) 855–877. doi:10.1080/08927022.2014.919497.

65. S. Seyler, O. Beckstein, Molecular dynamics trajectory for benchmarking MDAnalysis, online, 2017. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170. doi:10.6084/m9.figshare.5108170.

66. E. Lindahl, B. Hess, D. van der Spoel, Gromacs 3.0: A package for molecular simulation and trajectory analysis, J. Mol. Mod. 7 (2001) 306–317. URL: http://www.gromacs.org. doi:10.1007/s008940100045.

67. D. Spångberg, D. S. D. Larsson, D. van der Spoel, Trajectory NG: portable, compressed, general molecular dynamics trajectories, J Mol Model 17 (2011) 2669–85. doi:10.1007/s00894-010-0948-5.

68. D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, B. Towles, Millisecond-

scale molecular dynamics simulations on anton, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11. doi:10.1145/1654059.1654099.

69. D. E. Shaw, J. P. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, A. Forte, J. Gagliardo, G. Gill, B. Greskamp, C. R. Ho, D. J. Ierardi, L. Iserovich, J. S. Kuskin, R. H. Larson, T. Layman, L. Lee, A. K. Lerer, C. Li, D. Killebrew, K. M. Mackenzie, S. Y. Mok, M. A. Moraes, R. Mueller, L. J. Nociolo, J. L. Peticolas, T. Quan, D. Ramot, J. K. Salmon, D. P. Scarpazza, U. B. Schafer, N. Siddique, C. W. Snyder, J. Spengler, P. T. P. Tang, M. Theobald, H. Toma, B. Towles, B. Vitale, S. C. Wang, C. Young, Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 41–53. doi:10.1109/SC.2014.9.

70. R. Salomon-Ferrer, A. W. Götz, D. Poole, S. Le Grand, R. C. Walker, Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald, Journal of Chemical Theory and Computation 9 (2013) 3878–3888. URL: http://pubs.acs.org/doi/abs/10.1021/ct400314y. doi:10.1021/ct400314y.

71. J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, S. C. Glotzer, Strong scaling of general-purpose molecular dynamics simulations on gpus, Computer Physics Communications 192 (2015) 97–107. URL: http://www.sciencedirect.com/science/article/pii/S0010465515000867. doi:dx.doi.org/10.1016/j.cpc.2015.02.028.

72. A. Choudhary, W. keng Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, R. Latham, Scalable i/o and analytics, Journal of Physics: Conference Series 180 (2009).

73. S. W. Son, S. Sehrish, W. keng Liao, R. Oldfield, A. Choudhary, Reducing i/o variability using dynamic i/o path characterization in petascale storage systems,

Journal of Supercomputing 73 (2017) pp 2069–2097.

74. K.-W. Lin, J. Chou, S. Byna, K. W. and, Optimizing fast query performance on Lustre file system, in: SSDBM Proceedings of the 25th International Conference on Scientific and Statistical Database Management, Article No. 29, 2013.

75. G. Lockwood, 2017, What is so bad about POSIX I/O?, URL: `https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/`.

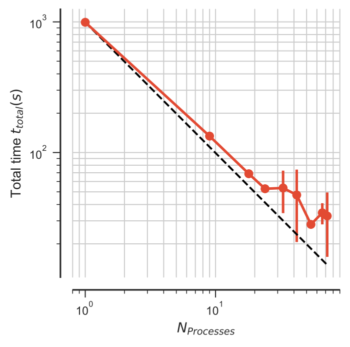
76. J. Mache, V. Lo, S. Garg, The impact of spatial layout of jobs on I/O hotspots in mesh networks, Journal of Parallel and Distributed Computing 65 (2005) 1190 – 1203. URL: `http://www.sciencedirect.com/science/article/pii/S0743731505001048`. doi:`10.1016/j.jpdc.2005.04.020`, design and Performance of Networks for Super-, Cluster-, and Grid-Computing Part I.

77. K. A. Brown, N. Jain, S. Matsuoka, M. Schulz, A. Bhatele, Interference between I/O and MPI traffic on fat-tree networks, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, ACM, New York, NY, USA, 2018, pp. 7:1–7:10. URL: `http://doi.acm.org/10.1145/3225058.3225144`. doi:`10.1145/3225058.3225144`.

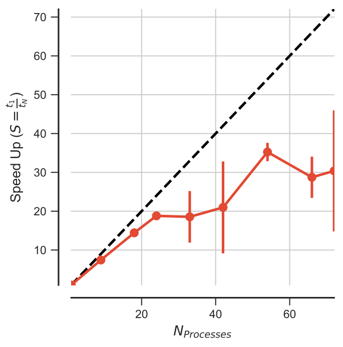
78. P. de Buyl, P. H. Colberg, F. Höfling, H5MD: A structured, efficient, and portable file format for molecular data, Computer Physics Communications 185 (2014) 1546 – 1553. doi:`10.1016/j.cpc.2014.01.018`.

Appendix A. Additional Data

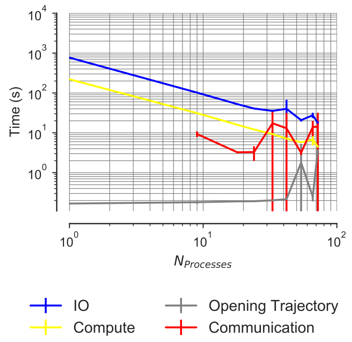
Figure A.11 shows performance of the RMSD task on *PSC Bridges*.



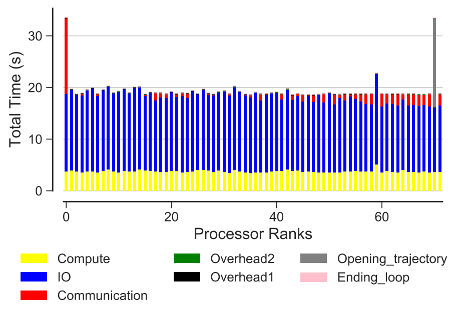
(a) Scaling total



(b) Speed-up



(c) Scaling for different components



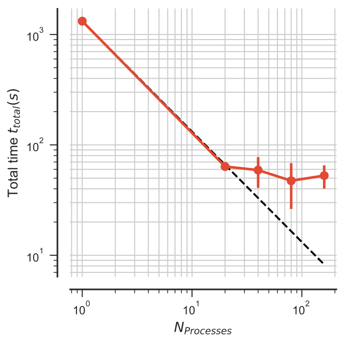
(d) Time comparison on different parts of the calculations per MPI rank (example)

Figure A.11: *PSC Bridges*: Performance of the RMSD task. Results are communicated back to rank 0. Five independent repeats were performed to collect statistics. (a-c) The error bars show standard deviation with respect to the mean. In serial, there is no communication and hence no data point is shown for $N = 1$ in (c). (d) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for definitions. These are data from one run of the five repeats. MPI ranks 0 and 70 are stragglers.

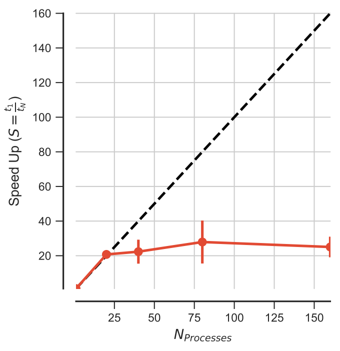
Figure A.12 shows performance of the RMSD task on *LSU SuperMIC*.

Figure A.13 shows comparison of the parallel efficiency of the RMSD task between different test cases on *SDSC Comet*, *PSC Bridges*, and *LSU SuperMIC*.

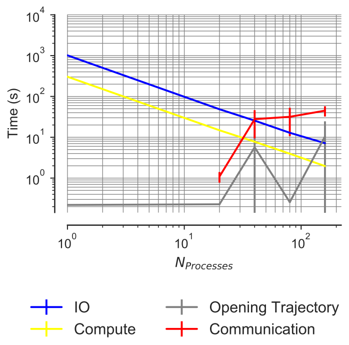
Figure A.14 shows how RMSD task scales with the increase in the number of cores when the trajectories are split using *Global Arrays* for communication compared to



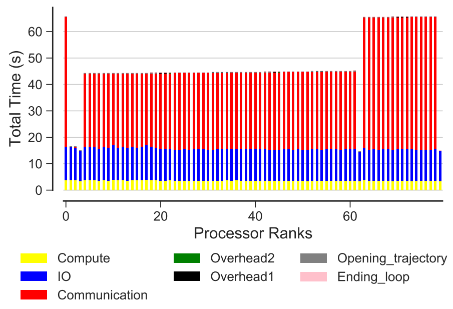
(a) Scaling total



(b) Speed-up



(c) Scaling for different components



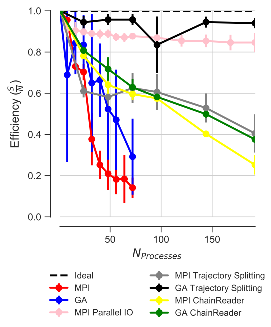
(d) Time comparison on different parts of the calculations per MPI rank (example)

Figure A.12: *LSU SuperMIC*: Performance of the RMSD task with MPI. Results are communicated back to rank 0. Five independent repeats were performed to collect statistics. (a-c) The error bars show standard deviation with respect to mean. In serial, there is no communication and hence the data points for $N = 1$ are not shown in (c). (d) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank; see Table 3 for definitions. These are data from one run of the five repeats.

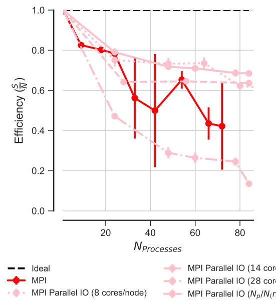
using MPI for communications on *LSU SuperMIC*.

Appendix B. Effect of $R_{\text{comp/comm}}$ on Performance

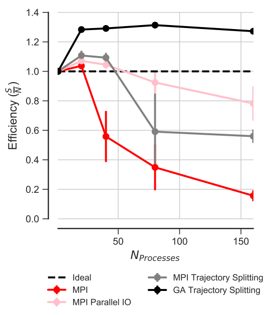
In Section 6.3, we improved scaling limitations due to read I/O by splitting the trajectory, but scaling remained far from ideal when MPI communication was used; the use of *Global Arrays* lead to better scaling (see Section 6.4) because the effective communication cost was reduced. Although we were not able to identify the reason for the better performance of *Global Arrays* (it still uses MPI as a communicator), the



(a) *SDSC Comet*

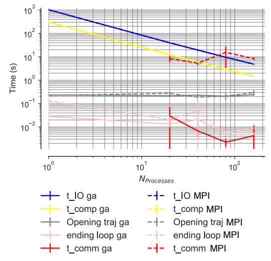


(b) *PSC Bridges*

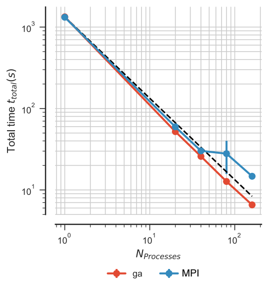


(c) *LSU SuperMIC*

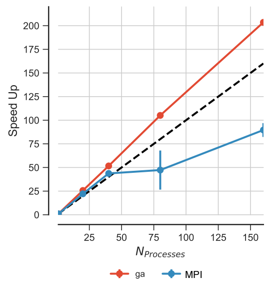
Figure A.13: Comparison of the parallel efficiency between different test cases on (a) *SDSC Comet* (data for "MPI Parallel IO" are only shown up to 192 cores for better comparison across different scenarios, see Fig. 6b for equivalent scaling data up to 384 cores), (b) *PSC Bridges*, and (c) *LSU SuperMIC*. Five repeats were performed to collect statistics and error bars show standard deviation with respect to mean.



(a) Scaling for different components



(b) Scaling total



(c) Speed-up

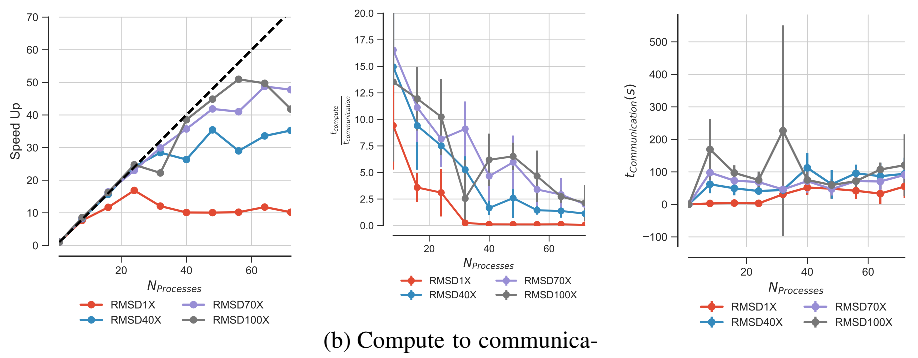
Figure A.14: *LSU SuperMIC*: Comparison of the performance of the RMSD task with subfiling and using either MPI ("MPI") or *Global Arrays* ("ga") for communication. For *Global Arrays*, all ranks update the global array (`ga_put()`) and rank 0 accesses the whole RMSD array through the global memory address (`ga_get()`). Five repeats were performed to collect statistics. (a) Compute and I/O scaling versus number of processes. (b) Total time scaling versus number of processes. (c) Speed-up. (a-c) The error bars show standard deviation with respect to mean.

results motivated an analysis in terms of the communication costs. In addition to the compute to I/O ratio $R_{\text{comp/IO}}$ discussed in Section 6.2 we defined another performance parameter called the compute to communication ratio $R_{\text{comp/comm}}$ (Eq. 8).

We analyzed the data for variable workloads (see Section 6.2) in terms of the $R_{\text{comp/comm}}$ ratio. The performance clearly depended on the $R_{\text{comp/comm}}$ ratio (Figure B.15). Performance improved with increasing $R_{\text{comp/comm}}$ ratios (Figure B.15b and B.15a) even if the communication time was larger (Figure B.15c). Although we still observed stragglers due to communication at larger $R_{\text{comp/comm}}$ ratios (70× RMSD and

100× RMSD), their effect on performance remained modest because the overall performance was dominated by the compute load. Evidently, as long as overall performance is dominated by a component such as compute that scales well, then performance problems with components such as communication will be masked and overall acceptable performance can still be achieved (Figures B.15a and B.15b).

Communication was usually not problematic within one node because of the shared memory environment. For less than 24 processes, i.e., a single compute node on *SDSC Comet*, the scaling was good and $R_{\text{comp/comm}} \gg 1$ for all RMSD loads (Figures B.15a and B.15b). However, beyond a single compute node (> 24 cores), scaling appeared to improve with increasing $R_{\text{comp/comm}}$ ratio while the communication overhead decreased in importance (Figures B.15a and B.15b).



(a) Speed-Up $S(N)$

(b) Compute to communication ratio $R_{\text{comp/comm}}$

(c) Communication time t_{comm}

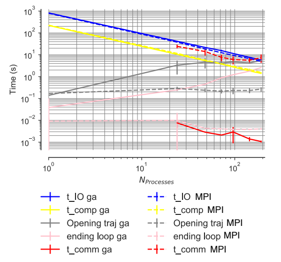
Figure B.15: Effect of the ratio of compute to communication time $R_{\text{comp/comm}}$ on scaling performance on *SDSC Comet*. (a) Scaling for different computational workloads. (Same as Figure 3a.) (b) Change in $R_{\text{comp/comm}}$ with the number of processes N for different workloads. (c) Comparison of communication time for different RMSD workloads. Five repeats were performed to collect statistics and error bars show standard deviation with respect to mean.

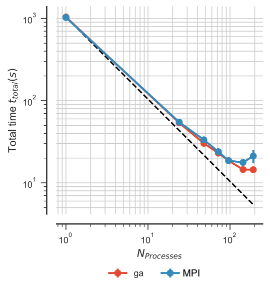
Appendix C. Performance of the ChainReader for Split Trajectories

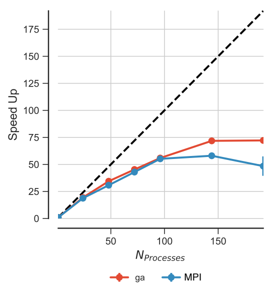
In section 6.3.1 we showed how subfiling (splitting the trajectories) would help to overcome I/O limitations and improve scaling. However, the number of trajectories may not necessarily be equal to the number of processes. For example, trajectories from MD simulations on supercomputers are often kept in small segments in individual files that need to be concatenated later to form a trajectory that can be analyzed with common tools. Such segments might be useful for subfiling but making sure that the number of processes is equal to the number of trajectory files will not always be feasible. *MDAnalysis* can transparently represent multiple trajectories as one virtual trajectory using the *ChainReader*. This feature is convenient for serial analysis when trajectories are maintained as segments. In the current implementation of ChainReader, each process opens all the trajectory segment files but I/O will only happen from a specific block of the trajectory specific to that process only.

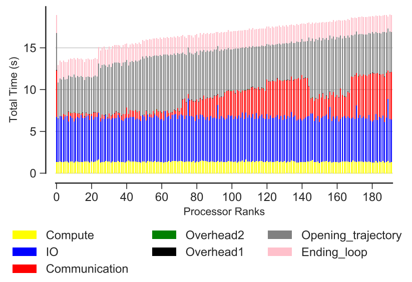
We wanted to test if the ChainReader would benefit from the gains measured for the subfiling approach. Specifically, we measured if the MPI-parallelized RMSD task (with N_p ranks) would benefit if the trajectory was split into $N_\mathrm{seg} = N_\mathrm{p}$ trajectory segments, corresponding to an ideal scenario.

In order to perform our experiments we had to work around an issue with the XTC format reader in *MDAnalysis* that was related to the XTC random-access functionality that the `MDAnalysis.coordinates.XTC.XTCReader` class provides: The Gromacs XTC format [66, 67] is a lossy-compression, XDR-based file format that was never designed for random access and the compressed format itself does not support fast random seeking. The `XTCReader` stores persistent offsets for trajectory frames to disk [18] in order to enable efficient access to random frames. These offsets will be generated automatically the first time a trajectory is opened and the offsets are stored in hidden `*.xtc_offsets.npz` files. The advantage of these persistent offset files is that after opening the trajectory for the first time, opening the same file will be very fast, and random access is immediately available. However, stored offsets can get out of sync with the trajectory they refer to. To prevent the use of stale offset data, trajectory file data (number of atoms, size of the file and last modification time) are also stored for validation. If any of these parameters change the offsets are recalculated. If the XTC changes but the offset file is not updated then the offset file can be detected as invalid. With ChainReader in parallel, each process opens all the trajectories because each process builds its own `MDAnalysis.Universe` data structure. If an invalid offset file is detected (perhaps because of wrong file modification timestamps across nodes), several processes might want to recalculate these parameters and rebuild the offset file, which can lead to a race condition. In order to avoid the race condition,


(a) Scaling for different components


(b) Scaling total


(c) Speed-up


(d) Time comparison of different parts of the calculations per MPI rank using ChainReader with MPI collective communications

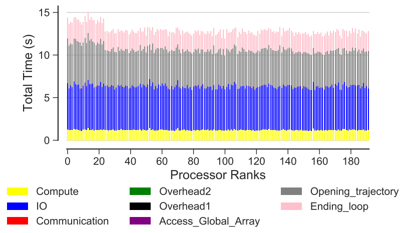

(e) Time comparison on different parts of the calculations per MPI rank using ChainReader using *Global Arrays*

Figure C.16: Comparison on the performance of the MDAnalysis ChainReader for the RMSD task on *SDSC Comet* when the trajectories are split; for the communication step either collective MPI ("MPI") or *Global Arrays* ("ga") was used. In case of *Global Arrays*, all ranks update the global array (`ga_put()`) and rank 0 accesses the whole RMSD array through the global memory address (`ga_get()`). Five repeats were performed to collect statistics. (a) Compute and I/O scaling versus number of processes. (b) Total time scaling versus number of processes. (c) Speed-up. (a-c) The error bars show standard deviation with respect to the mean. (d-e) Compute t_{comp}, read I/O $t_{\text{I/O}}$, communication t_{comm}, access to the whole global array by rank 0 $t_{\text{Access\_Global\_Array}}$, ending the for loop $t_{\text{end\_loop}}$, opening the trajectory $t_{\text{opening\_trajectory}}$, and overheads $t_{\text{overhead1}}$, $t_{\text{overhead2}}$ per MPI rank. (See Table 3 for the definitions.)

we removed this check from MDAnalysis for the purpose of the measurements presented here, but this comes at the price of not checking the validity of the offset files at all; future versions of MDAnalysis may lift this limitation. We obtained the results for the ChainReader with this modified version of *MDAnalysis* that eliminates the race condition by assuming that XTC index files are always valid.

Figure C.16 shows the results for performance of the ChainReader for the RMSD task using either collective MPI or Global Arrays (GA) for the communication step. With GA good strong scaling performance was observable up to 144 cores (Figure

C.16c); without GA, strong scaling plateaued for more than 92 cores. In both cases, strong scaling performance was worse than what was achieved when each MPI process was assigned its own trajectory segment as described in Section 6.3.1. The strong scaling performance did not suffer because of the computation and the read I/O because both t_{comp} and $t_{\text{I/O}}$ showed excellent strong scaling up to 196 cores (Figure C.16a). Instead the time for ending the `for` loop $t_{\text{end\_loop}}$, which includes the time for closing the trajectory file, and opening the trajectory $t_{\text{opening\_trajectory}}$ appeared to be the scaling bottleneck. These results differed from the subfiling results (section 6.3.1) where neither $t_{\text{end\_loop}}$ nor $t_{\text{opening\_trajectory}}$ limited scaling (Figures 5d and 5e).

Although we did not further investigate the deeper cause for the reduced scaling performance of the ChainReader, we speculate that the primary problem is related to each MPI rank having to open all trajectory files in their ChainReader instance even though they will only read from a small subset. For N_{p} ranks and N_{seg} file segments, in total, $N_p N_{\text{seg}}$ file opening/closing operations have to be performed. Each server that is part of a Lustre file system can only handle a limited number of I/O requests (read, write, stat, open, close, etc.) per second. A large number of such requests, from one or more users and one or more jobs, can lead to contention for storage resources. For $N_{\text{p}} = N_{\text{seg}} = 100$, the Lustre file system has to perform 10,000 of these operations almost simultaneously, which might degrade performance.